EFFICIENT SUPPORT FOR SPECULATIVE TASKING

BY

JAMES MURRAY TUCK, III

B.E., Vanderbilt University, 1999
M.S., University of Illinois at Urbana-Champaign, 2003

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois

Abstract

Improving application performance is a major challenge for computer architects. Two important reasons for it are the shift to multi-core architectures, which will no longer emphasize improvements in instruction-level parallelism, and long memory latencies. A versatile primitive for overcoming these obstacles is Speculative Tasking. With Speculative Tasking (ST), the outcome of a long or risky operation is assumed to be known, thereby allowing the execution of the following code section — potentially in parallel —by taking a hardware checkpoint and buffering the speculative state. Later, if it turns out that the assumption was incorrect, the hardware rolls back the whole section to the checkpoint and re-executes it transparently. ST has been studied in depth and has been shown to improve application performance. However, when applying ST to existing systems important problems and concerns become apparent. This dissertation considers three systems and three new problems facing Speculative Tasking.

The first system studied is a checkpointed processor that uses a task to speculate past cache misses rather than stalling; if the speculation is successful, the processor can potentially hide the latency of the memory access and boost Memory-Level Parallelism (MLP). However, the boost in MLP substantially increases the number of in-flight memory operations to the extent that conventional memory hierarchies are unsuited to support them — they need to be redesigned to support 1-2 orders of magnitude more outstanding misses. Yet, designing scalable MHAs is challenging: designs must minimize cache lock-up time and deliver high bandwidth while keeping the area consumption reasonable. Hence, a novel scalable MHA design for high-MLP processors is proposed that introduces two main innovations. First, it is *hierarchical*, with a small MSHR file per cache bank, and a larger MSHR file shared by all banks. Second, it uses a *Bloom filter* to reduce searches in the larger MSHR file. The result is a high-performance, area-efficient design. Compared to a state-of-the-art MHA on a high-MLP processor, the proposed design speeds-up some SPECint, SPECfp, and multiprogrammed workloads by a geometric mean of 32%, 50%, and 95%, respectively.

The second system studied is Speculative Multithreading (SM) on a Chip Multiprocessor (CMP). While it has the ability to speed-up hard-to-parallelize applications, the power ineffi-

ciency of aggressive speculation is a concern. To improve power efficiency, I note that not all the tasks that are running in such an environment are equally critical. To leverage these insights, a widely-applicable, novel task-criticality model for SM is developed for analyzing SM programs. Then, an architecture is proposed that (i) uses this model to analyze and predict the criticality of tasks in a SM application at run-time, and (ii) uses criticality to schedule tasks on a SM CMP for power efficient execution. Experiments with SPECint, SPECfp, and Olden show that CAP reduces $E \times D^2$ by 9%, 21%, and 23% on average respectively.

Finally, recent proposals for Speculative Tasking have called for *signatures* in hardware to accelerate memory disambiguation. The power of signatures lie in their ability to represent a set of addresses concisely while allowing for set operations directly on the signatures. As a result, they make costly set operations cheap to perform in hardware. To take full advantage of signatures, this thesis presents SoftBulk, a novel architecture that exposes signatures to software directly through the instruction set. Using SoftBulk, programs can collect information about their own memory access patterns and use that information for a variety of purposes including code optimization and debugging.

To Alyson:
Your continuous love and support made it all possible

Acknowledgments

Many people have made this work possible. First, I would like to thank my advisor, Josep Torrellas, for his kind advice and counsel during my graduate studies. There were many times that I was willing to give up on a research idea, however, his fortitude and perseverance encouraged me to keep working, and, as a result, I was invariably rewarded. I strive to have the same dedication and persistence in my own career.

This dissertation has been much improved through collaborations with all members of the I-ACOMA group. They have provided invaluable comments and feedback on my research, and I am very thankful for their active participation. Luis Ceze, in particular, has been an invaluable cohort on many adventures. His insights and energetic spirit have always spurred me on to the next research topic. I look forward to continuing our collaboration in the future. Wei Liu has also been a valued advisor and friend. Finally, Wonsun Ahn has dedicated considerable time and energy to the POSH and SoftBulk projects, and, for that, I am especially grateful. Karin Strauss, Radu Teodorescu, Paul Sack, and Brian Greskamp have also been especially helpful throughout graduate school. Whether it was just listening to an idea, going for ice cream, or making a cup of coffee they have left an indelible mark that I will always remember and appreciate. I also thank Sheila Clark for her help on many occasions. Her proficiency and thoroughness are truly inspiring.

In my early years in I-ACOMA, Jose Renau was instrumental in guiding me through the difficult process of learning a new simulation infrastructure, in showing me how to conduct large sets of experiments through effective automation, and in gaining a deeper understanding of the field. For a couple of years, I shared an office and an occasional racquet ball game with Chris Hughes and Milos Prvulovic, and I thank them for the exercise and their willingness to point out a bad (or good) idea when they heard one. Lee Baugh and I have shared many projects and many cups of tea since the beginning of my graduate studies. His advice and friendship have been an integral part of my graduate career and, in more than one situation, have made all the difference.

I also thank my Ph.D. committee for their time and due diligence in their service on my committee. I also thank them for their advice and support and for helping me move on to

the next step in my academic career.

Now, I must thank my family. Throughout my life, my parents have always supported and encouraged my academic endeavors. When I decided to attend high school thirty miles away in Birmingham, AL, at the Alabama School of Fine Arts, I was apprehensive about my own abilities. However, my parents never gave me the impression they had similar reservations, and their confidence inspired me to have confidence in myself. My pursuit in the Ph.D. was really no different. And this time around, it was not just my parents but my entire family that provided continual encouragement and support. Their belief in me has always encouraged me even when I found it hard to believe in myself. My brother, Ben, who attended medical school during my graduate studies was an indispensable ally not only for his good humor but because it helped to know I was not the only one working late nights and long hours for the sake of knowledge.

Last but not least, I thank my wife for all her support and encouragement over the last six years. It would double the length of this dissertation to fully express my gratitude. She is an ever present support and advisor in all my endeavors, and in many ways, this Ph.D. is hers as well.

Table of Contents

List of Tables

List of Figures

Chapter 1

Speculative Tasking

Improving application performance has long been an important design goal for computer architects. Fueled by relentless technology scaling, designers have leveraged ever increasing transistor budgets and faster transistors to build higher performing systems. However, advances in single-thread performance are increasingly difficult to attain due to the constraints of technology scaling, the design challenge of supporting a large number of in-flight instructions, and increasing cycle-times to main memory.

To live up to the challenge of improving application performance, architects have repeatedly turned to aggressive forms of speculative execution to make up the difference. Forms of speculative execution of interest in this dissertation are characterized by the following: (1) there is always a checkpoint to which execution can recover in the event of misspeculation, (2) speculatively executed instructions are retired instead of being buffered, and (3) speculative state is buffered until it is safe to propagate to the rest of the system. Architectural mechanisms that embody these three principles will be, for convenience, referred to as *Speculative Tasking* (ST) architectures. Figure 1.1(a) illustrates a simple case of speculative tasking with the three items identified above. Part (b) illustrates a more complex example of speculation spanning multiple threads.

Table 1.1 highlights several systems that embody the principles of ST, and the particular problem each one tackles. One use boosts memory-level parallelism (MLP) within a thread by allowing the processor to speculate past cache misses. It is also used in speculative parallelization, where a sequential application is automatically divided into multiple tasks that are executed in parallel. Similar to the previous case, Speculative Tasking has enabled aggressive compiler-directed optimizations.

However, ST still faces many problems. ST execution leads to increased pressure and bottlenecks on structures not directly involved in speculation; many forms of ST are energy inefficient; and, hooks that allow software to effectively leverage ST are missing. In the remainder of this chapter, ST will be considered in more detail to motivate the challenges listed above and to highlight the contributions made by this work.

Chip Level Arch.	Application Domain	Purpose	Speculative Tasking Architecture
Single Core	Single Threaded	Resource recycling	CPR [1], Cherry [50], KILO [23],
		Latency Tolerance, MLP	CFP [78], CAVA [10], Clear [40]
		Spec. Optimization	Software Spec [15, 57, 81]
Multiple Cores	Single Threaded	Speculative Multithreading & Parallelization	Multiscalar [76], Hydra [33], IACOMA [67] JRPM [16], STAMPede [79]
	Multi-threaded	Checkpointed proc.	CherryMP [39]
		Efficient Synch & Atomicity	SLE [63], Spec.Sync. [51], LogTM [97], TCC [35]
		Consistency	BulkSC [12],SWFMP [94]

Table 1.1: Overview of Speculative Tasking.

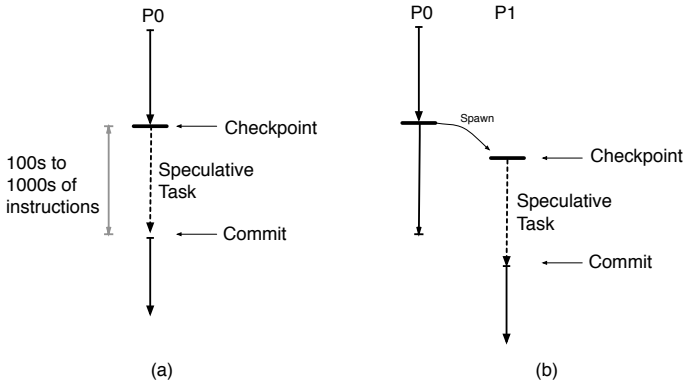


Figure 1.1: Speculative Tasking

1.1 Latency Tolerance and Early Resource Recycling

To increase instruction-level parallelism (ILP), processor designers typically grow the size of processor structures to support more in-flight instructions. However, large structures tend to be slow and lengthen cycle times. ST allows execution to proceed past long latency instructions that would otherwise stall the processor without actually buffering the instructions; consequently, structures can be small with low cycle times while still allowing for more in-flight instructions. Table 1.1 shows several systems that fall into this category.

CPR [1] and Out-of-order Commit [23] processors remove scalability bottlenecks from the I-window to substantially increase the number of in-flight instructions. They remove the ROB, relying on checkpointing (e.g., at low-confidence branches) to recover in case of misspeculation. CFP [78] frees the resources of a missing load and its dependent instructions without executing them. This allows the processor to continue fetching and executing

independent instructions. The un-executed instructions are buffered and executed when the data returns from memory.

Checkpoint-based value prediction schemes like CAVA [11] and CLEAR [40] checkpoint on a long-latency load miss, predict the value that the load will return, and continue execution using the prediction. Speculative instructions are allowed to retire. If the prediction is later shown to be correct, no rollback is necessary.

1.2 Speculative Multithreading and Synchronization

Due to power constraints of integrating hundreds of millions of transistors on a single chip, large monolithic processors are no longer favored by industry. Instead, many smaller processors are being integrated onto a single chip because they offer a better performance per watt design. In this genre, however, single thread performance is disadvantaged for a couple of notable reasons: (1) improvements to a single core are significantly constrained, and (2) automatic parallelization for general codes is very poor. Fortunately, ST offers several advantages for single-thread performance in the context of multi-core designs.

As detailed above, ST supports a large number of in-flight instructions while using modestly sized processor structures. This enables a smaller core, with few changes over an out-of-order execution engine, to attain high performance. The ST hardware can be utilized in many ways to speedup up the operation of a single-thread. Due to their architectural efficiency, ST cores are a good choice for a building block in multi-core systems.

Furthermore, ST can enhance the performance of individual threads in ways specific to multi-core systems. For the case of a single thread, ST has been shown as an effective technique to speculatively parallelize programs. In particular, Speculative Multithreading (SM) speeds-up a single thread on multiple processors by speculatively partitioning a program into threads and executing them in parallel. Since the end result must be the same as if the threads executed in sequential succession, hardware must track the order of threads as well as data and control dependences between threads.

Figure 1.1(b) illustrates this system. Note that the conditions for correct execution now span the execution of threads on multiple cores, and one thread's correct execution will depend on that of a predecessor thread. Speculative Multithreading is a powerful system because it can speculatively parallelize a program without proving it parallel in advance.

Speculative Tasking has also been applied to multithreaded programs in a few different ways. Speculative Lock Elision [63], Speculative Synchronization [51], and Hardware Transactional Memory [2, 14, 53, 64] can speedup some parallel programs by reducing lock

contention and synchronization overheads. They allow two threads to execute code speculatively that is protected by the same lock or synchronization primitive. If the two threads actually touch the same shared data, then one must rollback and re-execute, otherwise both are allowed to proceed unencumbered.

Finally, ST has also been used to enable strong consistency models with high performance [12, 94]. ST can enforce sequential consistency (SC) with high performance by restricting possible interleavings of instructions from different processors. Multiple stores will commit at once in a speculative task making them all visible at once. However, a speculative task may be re-executed if it violates SC conditions.

1.3 Speculative Parallelization and Optimization

Compilation support for ST has focused primarily on speculative parallelization [4, 16, 25, 47, 49, 59, 84, 92, 98] for multi-core architectures. The Multiscalar compiler [91] selects tasks by walking the Control Flow Graph (CFG) and accumulating basic blocks into tasks using a variety of heuristics. The task selection methodology for the Multiscalar compiler was recently revisited by Johnson *et al.* [37]. Instead of using a heuristic to collect basic blocks into tasks, the CFG is now annotated with weights and broken into tasks using a min-cut algorithm. These compilers assume special hardware for dispatching threads and, therefore, do not specify when a thread should be launched.

A number of compilers focus only on loops [25, 26, 85, 98]. In SPSM [26], loop iterations are selected by the compiler as speculative threads. An interesting part of the work is the use of the *fork* instruction that allows the compiler to specify when tasks begin executing. In addition, SPSM recognized the potential benefits from prefetching but proposed no techniques to exploit it. Du et al [25] recently presented a cost-driven compilation framework to statically determine which loops in a program deserve speculative parallelization. They compute a cost graph from the control flow and data dependence graphs and estimate the probability that misspeculation will occur along different paths in the graph. The cost graph, in addition to a set of criteria, determine which loops in a program deserve speculation.

Bhowmik and Franklin [4] built a framework for speculative multithreading on the SUIF-MachSUIF platform. Within this framework they considered dependence-based task selection algorithms and looked at thread spawning strategies. Like Multiscalar, they focus on compiling the whole program for speculation but allowed the compiler to specify a spawn location as in SPSM.

Some work has used dynamic information to improve selection of tasks for TLS [16, 49,

95]. JRPM [16] decomposes a Java program into threads dynamically using a hardware profiler called TEST. While the program runs in TEST, they identify important loops that will provide the most benefit due to speculative parallelization and recompile them with dynamic compilation support. Marcuello and Gonzalez [49] use profiling to identify tasks but are primarily interested in thread-spawning policies. Whaley and Kozyrakis [95] identify the profiler as a convenient and effective technique to improve task selection, and they show that simple profiling techniques can provide large performance gains. However, they only consider subroutine continuations as tasks, and they do not consider prefetching effects in their profiler.

Many other works have looked at optimizations for speculative threads. Chen *et al.* [17] calculated a probability for each points-to relationship that might exist for a pointer at a given point in the program. This probability can be used to determine whether a squash is likely to occur due to a memory carried dependence. Zhai *et al.* [98] were concerned with task selection but primarily for replacing dependences with synchronization and alleviating the associated synchronization overheads. Oplinger *et al.* [59] looked for the best places within an application to speculate. One important contribution was the use of value prediction to speculate past function calls.

Surprisingly few schemes have identified ST as beneficial for speculative optimizations within a traditional compiler [57, 81]. Recently, Neelakantam et al ([57]) showed how transactional commit and abort can be used to optimize Java codes. Unlike traditional speculative optimization, this framework does not require fix-up code when the speculative operation fails. In addition, aggressive optimizations can be applied within the atomic region at will. In a similar vein, Su and Lipasti ([81]) suggest guarded regions, similar to speculative tasks, can enable speculative optimizations for Java. In their scheme, optimizations are performed on the assumption that certain guard conditions must always remain true. Hardware monitors these guards at runtime, and if a guard condition is violated, the task rolls-back and executes non-optimized code. Similarly, Chen and Wu ([15]) suggest using TLS hardware for hot-path optimization and value prediction, allowing the hardware to recover from the cases when the optimization fails.

However, other single thread optimizations have been proposed that leverage speculation supports. Master/Slave Speculative Parallelization [100] optimizes the hot paths of a program and relies on speculative slave threads that execute the whole program to verify its optimized execution. Wu *et al.* ([96]) use a speculative thread to check the correctness of memoized code regions. While some dynamic optimization schemes have relied on checkpoint and undo, they have not leveraged a full compiler infrastructure and their optimizations were more limited in scope.

1.4 Efficient Support for Speculative Tasking

This remainder of this dissertation discusses three challenges facing some ST architectures and proposes architectural solutions.

1.4.1 Scalable Cache Miss Handling

Latency tolerant architectures like CAVA, Clear, and Out-of-order Commit drastically increase the number of outstanding misses that must be serviced by the L1 data cache. Not surprisingly, these microarchitectures require dramatic increases in Memory Level Parallelism (MLP) — broadly defined as the number of concurrent memory system accesses supported by the memory subsystem [18]. For example, one of these designs assumes support for up to 128 outstanding L1 misses at a time [78]. To reap the benefits of these microarchitectures, cache hierarchies have to be designed to support this level of MLP.

Current cache hierarchy designs are woefully unsuited to support this level of demand. Even in designs for high-end processors, the norm is for L1 caches to support only a very modest number of outstanding misses at a time. For example, Pentium 4 only supports 8 outstanding L1 misses at a time [6]. Unless the architecture that handles misses (i.e., the *Miss Handling Architecture* (MHA)) is redesigned to support 1-2 orders of magnitude more outstanding misses, there will be little gain to realize from the new microarchitectures.

First, it shows that state-of-the-art MHAs are unable to leverage the new high-MLP processor microarchitectures. Second, it proposes a novel, scalable MHA design for these microarchitectures that delivers the highest performance for a given area consumption. The proposed organization, called *Hierarchical*, introduces two main innovations that will be elaborated on further in Chapter 2. Finally, this work evaluates *Hierarchical* in the context of a high-MLP processor for some SPECint, SPECfp, and multiprogrammed workloads and show it to be the favored design.

1.4.2 Power and Energy Efficient Speculative Multithreading

While evaluations of SM on a CMP have generally shown good, if modest, speedups, an important concern has been the power inefficiency of aggressive speculation. Indeed, as more tasks are executed speculatively to deliver higher speedups, there is a higher chance of spending power on work that ultimately gets squashed. Wasting power is a very unattractive proposition, as power and energy consumption are currently major constraints in processor design.

Given the key importance of power issues, the design power-efficient SM systems are critical to their deployment and use. Previous work on this area by Renau *et al.* [66] focused on improving the energy efficiency of SM operations. In this work, the interaction between the tasks of an application are considered. Specifically, I make a key observation on the behavior of SM tasks: not all of the tasks that are running in an SM environment are equally critical for performance and power-efficient execution of the application — some are more critical than others.

To leverage this insight, two architectural features are needed. First, the CMP has to be able to assess the criticality of each task. Previous work on criticality analysis focused on instruction-level criticality [30, 43, 68, 89, 90]. While some of the ideas can be reused for tasks, the model and hardware implementation required are substantially different. Second, the CMP must be able to schedule tasks in a power efficient way in accordance with their criticality. This can be supported with Dynamic Voltage and Frequency Scaling (DVFS) on a per-core basis.

Based on these observations and needs, I propose *CAP*, a novel architecture that (i) analyzes and predicts the criticality of tasks in a SM application at run-time, and (ii) uses criticality to schedule tasks on a SM CMP with per-core DVFS for power-efficient execution. Critical tasks are scheduled on fast cores, while non-critical ones run on slower ones.

1.4.3 Effective Runtime Disambiguation for Speculative Optimizations

Code optimizers [57, 81] have identified ST as a valuable tool for compiler directed speculative optimization. However, these schemes have mostly focused on hot-path optimizations that allow the removal checks for various runtime errors, especially those found in modern languages like Java. Speculative optimizations that rely on explicit runtime disambiguation, on the other hand, have not been studied. This may be because a suitable hardware primitive that provided efficient disambiguation support was not available. However, *signatures* ([14]) are an effective means for storing and disambiguating sets of addresses and can be used to provide efficient runtime disambiguation in support of speculative optimizations.

I propose SoftBulk, an architecture for exposing signatures and bulk operations directly to software through the instruction set architecture (ISA). SoftBulk makes two key additions to the architecture: (i) instructions for accumulating addresses into signatures and for operating on them efficiently, and (ii) Bulk Register Files that are part of the architectural state of the processor. Using SoftBulk, programs can collect information about their own memory access patterns and use that information in many ways.

Purpose	Challenge	Solution
Latency tolerance	Support for many outstanding misses	Scalable Miss Handling [86]
Energy efficiency	Reduce energy cost of spec. parallelization	Criticality Analysis for S.M. [87]
Software primitives for optimization	Expose disambiguation primitives to software	SoftBulk

Table 1.2: Challenges and contributions.

To show the potential for SoftBulk, it is applied to function memoization. Memoization is a way of caching the results of a computation so that it need not be repeated–in this sense, it is a kind of dynamic redundancy elimination. For arbitrary functions which may have implicit memory inputs or produce side effects, memoization is usually not possible. However, SoftBulk allows a memoization algorithm for an arbitrary function in C/C++. The algorithm is called Signature Enhanced Memoization, or MemoiSE for short. MemoiSE leverages SoftBulk to record which memory locations are read and written into a signature and performs disambiguation on that signature. As a result, MemoiSE can easily test whether implicit inputs or side effects have been changed since the memoized call.

1.5 Summary

ST has been explored for both uni-core and multi-core systems. However, ST still faces many interesting problems. The remainder of this dissertation focuses on the three problems outlined in the previous section and shown in Table 1.2. The following chapters are organized as follows: Chapter 2 details the design of the Scalable Miss Handling Architecture for high MLP; Chapter 3 describes novel support for power-efficient SM; and Chapter 4 describes SoftBulk, a powerful architecture and interface for software directed disambiguation.

Chapter 2

Scalable Miss Handling Architectures

A flurry of recent proposals for novel superscalar microarchitectures claim to support very high numbers of in-flight instructions and, as a result, substantially boost performance [1, 11, 23, 40, 55, 78, 99]. These microarchitectures typically rely on processor checkpointing, long speculative execution, and sometimes even speculative retirement. They often seek to overlap cache misses by using predicted values for the missing data, by buffering away missing loads and their dependents, or by temporarily using an invalid token in lieu of the missing data. Examples of such microarchitectures include Runahead [55], CPR [1], Out-of-order Commit [23], CAVA [11], CLEAR [40], and CFP [78].

Not surprisingly, these microarchitectures require dramatic increases in Memory Level Parallelism (MLP) — broadly defined as the number of concurrent memory system accesses supported by the memory subsystem [18]. For example, one of these designs assumes support for up to 128 outstanding L1 misses at a time [78]. To reap the benefits of these microarchitectures, cache hierarchies have to be designed to support this level of MLP.

Current cache hierarchy designs are woefully unsuited to support this level of demand. Even in designs for high-end processors, the norm is for L1 caches to support only a very modest number of outstanding misses at a time. For example, Pentium 4 only supports 8 outstanding L1 misses at a time [6]. Unless the architecture that handles misses (i.e., the *Miss Handling Architecture* (MHA)) is redesigned to support 1-2 orders of magnitude more outstanding misses, there will be little gain to realize from the new microarchitectures.

A brute-force approach to increasing the resources currently devoted to handling misses is not the best route. The key hardware structure in the MHA is the Miss Status Holding Register (MSHR), which holds information on all the outstanding misses for a given cache line [28, 42]. Supporting many, highly-associative MSHRs in a unified structure may end up causing a bandwidth bottleneck. Alternatively, if the designer tries to ensure high bandwidth by extensively banking the structure, the MHA may run out of MSHRs in a bank, causing cache bank lock-up and eventual processor stall. In all cases, since the MHA is located close to the processor core, it is desirable to use the chip area efficiently.

To satisfy the requirements of high bandwidth and low lock-up time in an area-efficient

manner, this work presents a new, scalable MHA design for the L1 cache. More specifically, it makes the following three contributions.

First, it shows that state-of-the-art MHAs are unable to leverage the new high-MLP processor microarchitectures.

Second, it proposes a novel, scalable MHA design for these microarchitectures that delivers the highest performance for a given area consumption. The proposed organization, called *Hierarchical*, introduces two main innovations. First, it is a *hierarchical design*, composed of a small per-cache-bank MSHR file, and a larger MSHR file shared by all the cache banks. The per-bank files provide high bandwidth, while the shared one minimizes cache lock-up in the presence of cross-bank access imbalances in an area-efficient manner. The second innovation is a *Bloom filter* that eliminates the large majority of unnecessary accesses to the shared MSHR file, therefore removing a possible bottleneck.

Third, *Hierarchical* is evaluated in the context of a high-MLP processor for some SPECint, SPECfp, and multiprogrammed workloads. Compared to a state-of-the-art MHA similar to that of Pentium 4, *Hierarchical* speeds-up the workloads by a geometric mean of 32%, 50%, and 95%, respectively. Also, *Hierarchical* is compared to two intuitive extrapolations of current MHA designs, namely a large monolithic MSHR file and a large banked MSHR file. For the same area consumption, *Hierarchical* is faster by a geometric mean of 1-18% and 10-21%, respectively. Finally, *Hierarchical* performs very close to an unlimited-size, ideal MHA.

This chapter is organized as follows: Section 2.1 presents a background and motivation; Section 2.2 assesses the new MHA challenges; Sections 2.3 and 2.4 present the MHA design and its implementation; and Sections 2.5 and 2.6 evaluate the design.

2.1 Background and Motivation

2.1.1 Miss Handling Architectures (MHAs)

The *Miss Handling Architecture (MHA)* is the logic needed to support outstanding misses in a cache. Kroft [42] proposed the first MHA that enabled a lock-up free cache (one that does not block on a miss) and supported multiple outstanding misses at a time. To support a miss, he introduced a Miss Information/Status Holding Register (MSHR). An MSHR stored the address requested and the request size and type, together with other information. Kroft organized the MSHRs into an MSHR file accessed after the L1 detects a miss (Figure 2.1(a)). He also described how a store miss buffers its data so that it can be *forwarded* to a subsequent load miss before the full line is obtained from main memory.

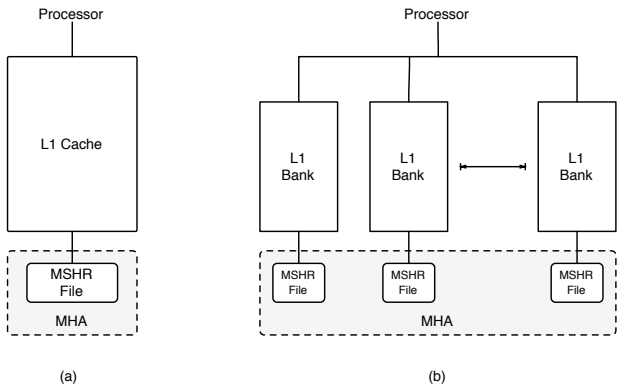


Figure 2.1: Examples of Miss Handling Architectures (MHAs).

Scheurich and Dubois [70] described an MHA for lock-up free caches in multiprocessors. Later, Sohi and Franklin [77] evaluated the bandwidth advantages of using cache banking in non-blocking caches. They used a design where each cache bank has its own MSHR file (Figure 2.1(b)), but did not discuss the MSHR itself.

A cache miss on a line is *primary* if there is currently no outstanding miss on the line and, therefore, a new MSHR needs to be allocated. A miss is *secondary* if there is already a pending miss on the line. In this case, the existing MSHR for the line can be augmented to record the new miss, and no request is issued to memory. In this case, the MSHR for a line keeps information for all outstanding misses on the line. For each miss, it contains a *subentry* (in contrast to an *entry*, which is the MSHR itself). Among other information, a subentry for a read miss contains the ID of the register that should receive the data; for a write miss, it contains the data itself or a pointer to a buffer with the data. Once an MHA exhausts its MSHRs or subentries, it *locks-up* the cache (or the corresponding cache bank). From then on, the cache or cache bank rejects further requests from the processor. This may eventually lead to a processor stall.

Farkas and Jouppi [28] examined *Implicitly*- and *Explicitly*-addressed MSHR organizations for read misses. In the Implicit one, the MSHR has a pre-allocated subentry for each word in the line. In the Explicit one, any of the (fewer) subentries in the MSHR can be used by any miss on the line. However, a subentry also needs to record the line offset of the miss it handles.

Current MHA designs are limited. For example, Pentium 4's L1 only supports 8 outstanding misses at a time [6] — as communicated to us by one of the designers, this includes both primary and secondary misses.

2.1.2 Microarchitectures for High MLP

Many proposed techniques to boost superscalar performance significantly increase MLP requirements. Among these techniques, there are traditional ones such as prefetching, multithreading, and other techniques discussed in [18]. Recently, however, there have been many proposals for novel processor microarchitectures that substantially increase the number of in-flight instructions [1, 11, 23, 40, 50, 55, 78, 99]. They typically leverage state checkpointing and, sometimes, retirement of speculative instructions. Unsurprisingly, they also *dramatically* increase MLP requirements (e.g., [78] assumes support for 128 outstanding L1 misses).

One of them, Runahead execution [55], checkpoints the processor and retires a missing load, marking the destination register as invalid. The instruction window is unblocked and execution proceeds, prefetching data into the cache. When the load completes, execution rolls back to the checkpoint. A related scheme by Zhou and Conte [99] uses value prediction on missing loads to continue execution (no checkpoint is made) and re-executes everything on load completion.

Checkpoint-based value prediction schemes like CAVA [11] and CLEAR [40] checkpoint on a long-latency load miss, predict the value that the load will return, and continue execution using the prediction. Speculative instructions are allowed to retire. If the prediction is later shown to be correct, no rollback is necessary.

CPR [1] and Out-of-order Commit [23] processors remove scalability bottlenecks from the I-window to substantially increase the number of in-flight instructions. They remove the ROB, relying on checkpointing (e.g., at low-confidence branches) to recover in case of misspeculation. CFP [78] frees the resources of a missing load and its dependent instructions without executing them. This allows the processor to continue fetching and executing independent instructions. The un-executed instructions are buffered and executed when the data returns from memory.

2.1.3 Vector MHAs

Vector machines have MHAs that differ markedly from the ones considered here. The reason is two fold. First, classical vector machines have memory systems that return misses in FIFO order. As a result, the MSHR file does not need to support associative searches and can be a simple, large FIFO file [22] — e.g., supporting 384 outstanding misses in the Cray SV1. Superscalar machines cannot afford such expensive memory systems and, therefore, need associative, more complex MHAs. The second difference is that many vector machines, such as the Cray SV1, have one-word lines, which simplifies the MHA substantially. Even

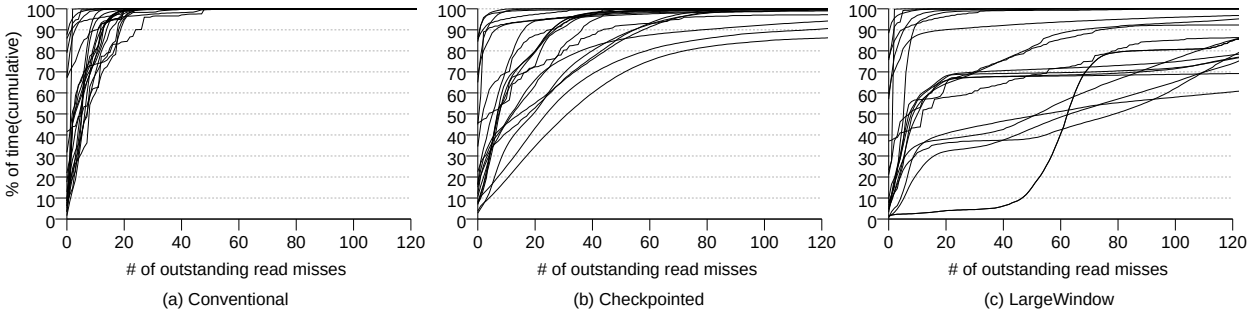


Figure 2.2: Number of oustanding L1 read misses at a time for each processor. Shown are *Conventional* (a), *Checkpointed* (b) and *LargeWindow* (c).

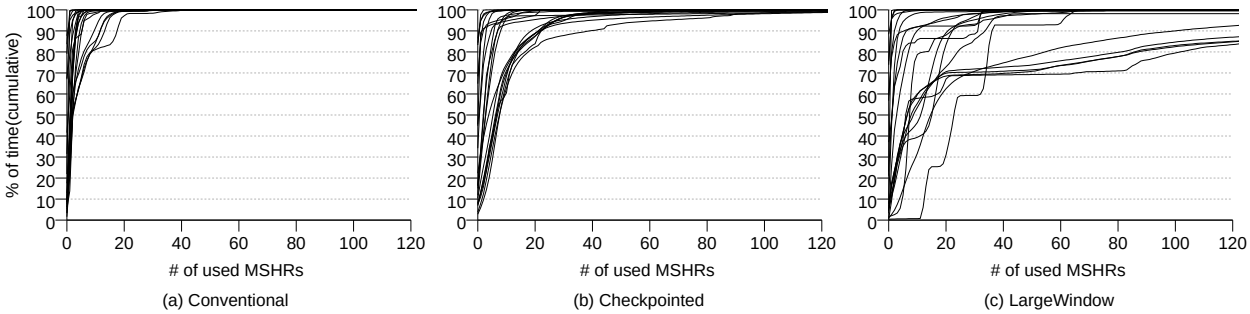


Figure 2.3: Number of L1 MSHR entries in use at a time for each processor. Shown are *Conventional* (a), *Checkpointed* (b) and *LargeWindow* (c). Only read misses are considered.

in vector designs that use multi-word lines, such as the Cray X1 [27], the use of vector loads/stores (and vector prefetches) enables a simpler MHA design targeted to relatively few secondary misses. New MSHR designs have to support many secondary misses, as will be shown later.

2.1.4 Why Not Reuse the Load/Store Queue State?

The microarchitectures of Section 2.1.2 generate a large number of concurrent memory system accesses. These accesses need support at two different levels, namely at the load/store queue (LSQ) and at the cache hierarchy level. First, they need an LSQ that provides efficient address disambiguation and forwarding. Second, those that miss somewhere in the cache hierarchy need an MHA that efficiently handles many outstanding misses. While previous work has proposed solutions for scalable LSQs [32, 60, 72], the problem remains unexplored at the MHA level.

It is possible to conceive a design where the MHA is kept to a minimum by leveraging

the LSQ state. Specifically, a simple MSHR is allocated on a primary miss and keeps no additional state on secondary misses — the LSQ entries corresponding to the secondary misses can keep a pointer to the corresponding MSHR. When the data arrives from memory, the LSQ is searched with the MSHR ID and all the relevant LSQ entries are satisfied.

However, this is not a good design for the advanced microarchitectures described.

First, it induces global searches in the large LSQ. Recall that scalable LSQ proposals provide efficient search from the *processor-side*. The processor uses the word address to search. In the approach discussed, LSQs would also have to be searched from the *cache-side*, when a miss completes. This would involve a search using the MSHR ID or the line address, which (unless the LSQ is redesigned) would induce a costly global LSQ search. Such search is eliminated if the MHA is enhanced with subentry pointer information.

Second, some of these novel microarchitectures speculatively *retire* instructions and, therefore, deallocate their LSQ entries [11, 40]. Consequently, the MHA cannot rely on information in LSQ entries because, by the time the miss completes, the entries may be gone.

Finally, LSQs are timing-critical structures. It is best not to augment them with additional pointer information or support for additional types of searches. In fact, the best strategy is likely the one that *avoids restricting* their design at all.

Consequently, primary and secondary miss information are kept in the MHA and rely on no specific LSQ design.

2.2 Requirements for the New Miss Handling Architectures (MHA)

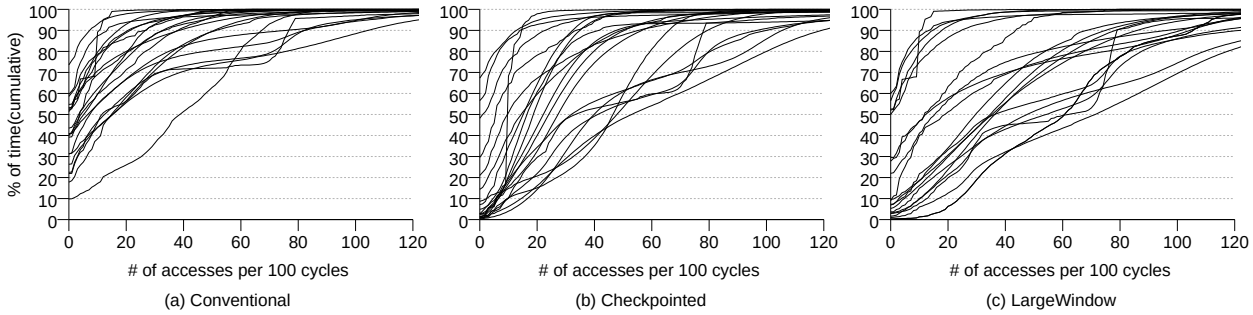


Figure 2.4: Bandwidth required from the MHA. Shown are *Conventional* (a), *Checkpointed* (b) and *LargeWindow* (c).

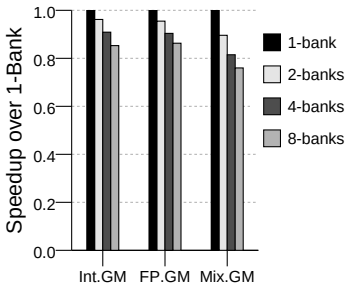


Figure 2.5: Performance impact of varying the number of MHA banks. Banks have unlimited bandwidth.

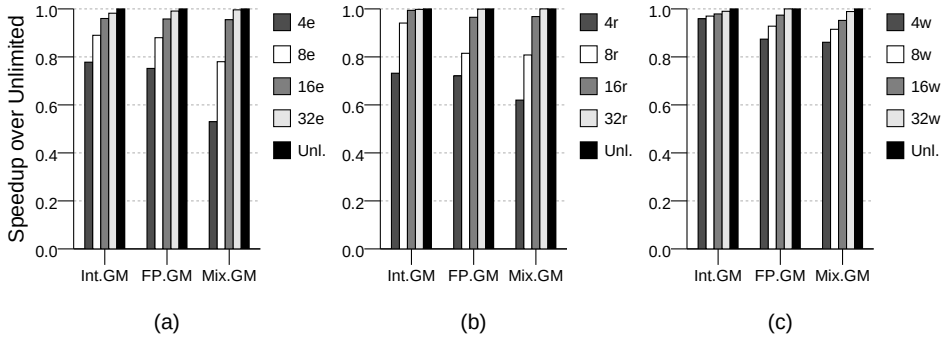


Figure 2.6: Performance impact of varying an MHA design parameter. The figure shows the number of entries (a), read subentries (b), and write subentries (c). In each case, we vary one parameter and keep the other two unlimited. The MHA has unlimited bandwidth.

The MLP requirements of the new microarchitectures described put major pressure on the cache hierarchy, especially at the L1 level. To handle it, known techniques, such as banking the L1 and making it set-associative, are used. However, a lesser known yet acute problem remains, namely that the MHA in the L1 is asked to store substantially more information and sustain a higher bandwidth than in conventional designs. This is a new challenge.

In this section, this challenge is assessed by comparing three processors: *Conventional*, *Checkpointed*, and *LargeWindow*. *Conventional* is a 5-issue, 2-context SMT processor slightly more aggressive than current ones. *Checkpointed* is *Conventional* enhanced with checkpoint-based value prediction like CAVA [11]. On L2 misses, it checkpoints, predicts the value and continues, retiring speculative instructions. *LargeWindow* is *Conventional* with an unrealistic 512-entry instruction window and 2048-entry ROB. All processors have a 32-Kbyte 2-way L1 organized in 8 banks. The bandwidth of the memory bus is 15GB/s. The rest of the

parameters are in Table 2.5 and will be discussed in section 2.5. For this section only, an MHA with ideal characteristics is modeled: unless otherwise indicated, it has an unlimited number of entries (MSHRs), and an unlimited number of subentries per MSHR. A summary of this section plus additional details are available in [13].

2.2.1 The New MHAs Need High Capacity

Figure 2.2 shows the distribution of the number of outstanding L1 read misses at a time. It shows the distributions for the three processors. Each line corresponds to one workload, which can be either one or two concurrent applications from SPECint2000 and SPECfp2000 — the workloads will be discussed in section 2.5.2.

For *Conventional*, most workloads have 16 or fewer outstanding load misses 90% of the time. These requirements are roughly on a par with the MHA of state-of-the-art superscalars. On the other hand, *Checkpointed* and *LargeWindow* are a stark contrast, with some workloads sustaining 120 outstanding load misses for a significant fraction of the time.

The misses in Figure 2.2 include both primary and secondary misses. Suppose now that a single MSHR holds the state of all the misses to the same L1 line. Figure 2.3 redraws the data showing the number of MSHRs in use at a time. An L1 line size of 64 bytes is used.

Compared to the previous figure, the distributions move to the upper left corner. The requirements of *Conventional* are few. For most workloads, 8 MSHRs are enough for 95% of the time. However, *Checkpointed* and *LargeWindow* have a greater demand for entries. *Checkpointed* needs about 32 MSHRs to have a similar coverage for most workloads. *LargeWindow* needs even more.

2.2.2 The New MHAs Need High Bandwidth

The MHA is accessed at two different times. First, when an L1 miss is declared, the MHA is read to see if it contains an MSHR for the accessed line. In addition, at this time, if the L1 miss cannot be satisfied by data forwarded from the MHA, the MHA is updated to record the miss. Second, when the requested line arrives from the L2, the MHA is accessed again to pull information from the corresponding MSHR and then clear the MSHR. Based on the assumed width of the MHA ports in this experiment, one access per write subentry or per four read subentries are assumed.

The number of MHA accesses for both read and write misses during 100-cycle intervals are computed for *Conventional*, *Checkpointed*, and *LargeWindow*. Figure 2.4 shows the distribution of the number of accesses per interval. For *Conventional*, many workloads have

at most 40 accesses per interval about 90% of the time. For *Checkpointed*, the number of accesses to reach 90% of the time is often around 60. For *LargeWindow*, still more accesses are required to reach 90%. Overall, new MHAs need to have higher bandwidth than current ones.

2.2.3 Banked MHAs May Suffer From Access Imbalance Lock-ups

To increase MHA bandwidth, the MHA is banked like the L1 cache (Figure 2.1(b)). However, since the number of MSHRs in the MHA is limited due to area constraints, heavy banking may be counter-productive: if the use of the different MHA banks is imbalanced, one of them may fill up. If this happens, the corresponding L1 bank locks-up; it rejects any further requests from the processor to the L1 bank. Eventually, the processor may stall. This problem is analogous to a cache bank access conflict in a banked L1 [38], except that a "conflict" in a fully-associative MHA bank may last for as long as a memory access time.

To assess this problem, an MHA with 16 MSHRs is used (unlimited number of subentries) and experiments are run grouping them into different numbers of banks: 1 bank of 16 entries, 2 of 8, 4 of 4, or 8 banks of 2 MSHRs. In all cases, a bank is fully associative and has no bandwidth limitations. The L1 has 8 banks.

Figure 2.5 shows the resulting performance of *Checkpointed*. The figure has three sets of bars, which correspond to the geometric mean of the SPECint2000 applications used (*Int.GM*), SPECfp2000 (*FP.GM*), and multiprogrammed mixes of both (*Mix.GM*). The figure shows that, as the number of MHA banks is increased, the performance decreases. Since a bank has unlimited bandwidth, contention never causes stalls. Stalls occur only if an MHA bank is full. Therefore, the designer needs to be wary of banking the new MHAs too much since, if each MHA bank has modest capacity, access imbalance may cause cache bank lock-up.

2.2.4 The New MHAs Need Many Entries, Read Subentries, and Write Subentries

L1 misses can be either primary or secondary, and be caused by reads or writes. For each case, the MHA needs different support. For primary misses, it needs MSHR entries; for secondary ones, it needs subentries. The latter typically need different designs for reads and for writes. To assess the needs in number of entries, read subentries, and write subentries, a single-bank MHA with unlimited bandwidth in *Checkpointed* is used. One parameter is

Workload	Number of used MSHRs (%)		
	Read Sub Only	Write Sub Only	Read+Write Sub
Int Avg.	67.8	26.5	5.7
FP Avg.	85.3	10.8	3.9
Mix Avg.	85.1	10.9	4.1
Total Avg.	79.4	16.1	4.6

Table 2.1: Usage of MSHRs in *Checkpointed*.

varied while keeping the other two unlimited. If the varying dimension runs out of space, the L1 refuses further processor requests until space opens up.

In Figure 2.6(a), the number of MSHR entries are varied. These workloads benefit significantly by going from 8 to 16 MSHRs, and to a lesser extent by going from 16 to 32 MSHRs. In Figure 2.6(b), the number of read subentries per MSHR are varied. Secondary read misses are frequent, and supporting less than 16-32 read subentries hurts performance. Finally, in Figure 2.6(c), the number of write subentries per MSHR are varied. Secondary write misses are also important, and around 16-32 write subentries are needed.

An additional insight is that individual MSHRs typically need read subentries or write subentries, but less frequently both kinds. This data is shown in Table 2.1 for *Checkpointed* running with an unlimited MHA. This behavior is due to the spatial locality of read and write misses. This will be leveraged in section 2.4.5.

2.3 An MHA for High MLP

Given these requirements for the MHA, it follows that current designs such as the one in Pentium 4, which only support 8 outstanding misses (of primary or secondary type) at a time [6], will be insufficient. One solution is to build a large, associative, centralized MSHR file. This design is called *Unified* (Table 2.2). It has a high capacity and does not cause L1 bank lock-up due to access imbalance (section 2.2.3). However, its centralization limits its bandwidth.

Another solution is a large, associative MSHR file that is banked like the L1 cache. This design is called *Banked*. It has higher bandwidth than *Unified*. However, under program behavior with access imbalance, one of the banks may fill up, causing L1 bank lock-up.

These shortcomings are addressed with the proposed two-level MHA design. It is called *Hierarchical* (Table 2.2). It has a small per-bank MSHR file called *Dedicated* and a larger

Design	Characteristics
Current	8 outstanding L1 misses like Pentium 4
Unified	Large centralized associative MSHR file with many entries and subentries
Banked	Large associative MSHR file with many entries and subentries, banked like L1
Hierarchical	Two-level design with a small per-bank MSHR file (*Dedicated*) and a larger MSHR file shared by all banks (*Shared*)

Table 2.2: MHA designs considered.

Goal	Proposed Solution
Minimize L1 lock-up while using MHA area efficiently	– *Shared* file – Fewer subentries in *Dedicated* files; more subentries in *Shared* file.
High bandwidth	– Per-bank *Dedicated* file – Allocate new entries always in *Dedicated* file: If entry is in MHA, locality typically ensures that it is found in *Dedicated* file – Bloom filter for *Shared* file (no false negatives, few false positives): If entry is not in MHA, filter typically averts access to *Shared* file

Table 2.3: Innovations in the proposed hierarchical MHA.

MSHR file shared by all banks called *Shared*. The Dedicated and Shared files have exclusive contents and are accessed sequentially. Entries that overflow from a Dedicated file are collected in the Shared file. Figure 2.7 shows the design. Table 2.3 lists this scheme's key innovations, which are considered next.

2.3.1 Minimize L1 Lock-up With Area Efficiency

A key goal of any MHA is to minimize the time during which it is out of MSHR file entries or subentries, while using the area efficiently. The latter is important because the MHA uses area close to the processor core. Consequently, MHA designs that minimize L1 lock-up time per unit area are preferred.

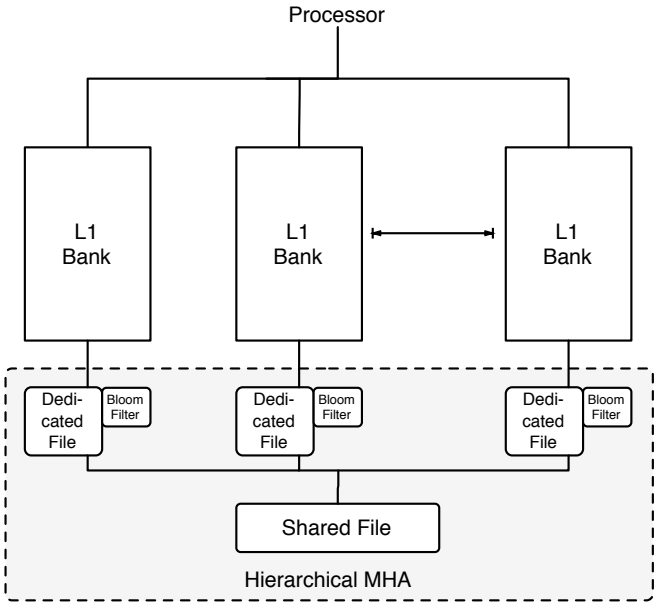


Figure 2.7: Proposed hierarchical MHA.

Banked does not use area very efficiently, since its capacity is divided into banks. If there is significant access imbalance across banks, one MHA bank can fill up and lock-up the cache bank.

Hierarchical uses area efficiently for two reasons (Table 2.3). First, it provides shared capacity in the Shared file — for a given need, one shared structure is more efficient than several private ones of the same total combined size, when there is access imbalance across structures. Second, a common reason for lock-up is that a few MSHRs need more subentries than the others and run out of them. In *Hierarchical*, rather than giving a high number of subentries to all the MSHRs, the Dedicated files are designed with fewer subentries. This enables area savings. If an entry in a Dedicated file runs out of subentries, it is displaced to the Shared file.

2.3.2 High Bandwidth

The other key goal of any MHA is to deliver high bandwidth. *Unified*, due to its centralization, has modest bandwidth.

Hierarchical attains higher bandwidth through three techniques (Table 2.3). First, it provides a per-bank Dedicated file. Second, when a new MSHR is needed, it is always allocated in the Dedicated file. If there is no free entry in the Dedicated file, one is displaced to the Shared file to open up space. As victim the entry that was inserted first in the Dedicated file (FIFO policy) is selected. Due to the spatial locality of cache misses, a primary L1 cache miss is often quickly followed by a series of secondary misses on the same line. With such policies, these secondary misses are likely to hit in the Dedicated file. The result is higher bandwidth and lower latency.

Finally, each bank also includes a small Bloom filter. It hashes the addresses of all the MSHR entries that were displaced from that bank's Dedicated file to the Shared file. The filter in a bank is accessed at the same time as the Dedicated file, and takes the same time to respond. When the Dedicated file misses, the filter indicates whether or not the requested entry may be in the Shared file. If the filter says "no", since a Bloom filter has *no false negatives*, the Shared file is not accessed. This saves a very large number of unneeded accesses to the Shared file, enhancing the MHA bandwidth. If the filter says "yes", the Shared file is accessed, although there may be a small number of false positives.

2.4 Implementation

This section describes several implementation aspects of the *Hierarchical* proposal: the overall organization and timing, the Bloom filter, the Dedicated file replacement algorithm, the implementation complexity, and the MSHR organizations.

2.4.1 Overall Organization and Timing

Each Dedicated file is fully pipelined and has a single read/write port. The Dedicated file and the filter in the same bank are accessed in parallel (Figure 2.7). In most cases, the outcome is either a hit in the Dedicated file or a miss in both the Dedicated file and filter. The first case is a secondary L1 miss intercepted by the Dedicated file. The second case is a primary L1 miss, in which case the Dedicated file allocates an entry and a request is sent to L2.

When a cache miss hits in the Bloom filter and misses in the Dedicated file, the Shared

file is accessed. Since this case is less frequent, the Shared file is a single-ported, slow and large structure. It is highly associative and unpipelined. Each entry has many subentries to support many outstanding secondary misses per line.

2.4.2 Bloom Filter

A Bloom filter without false negatives is employed, although some false positives can occur. Some Bloom filter designs require that the filter be periodically purged, so that aliasing does not create too many false positives. However, retraining the filter during operation could lead to false negatives. Consequently, a counter-based Bloom filter design is chosen that is similar to the counter array in [54] which requires no purging. Every time that an entry is displaced from the corresponding Dedicated to the Shared file, a set of counters are incremented to add the address to the filter. The same counters are decremented when the entry is deallocated from the Shared file. The counters to increment or decrement are determined by several bit-fields in the line address. Finally, an access hits in the filter if all the counters corresponding to the address being checked are non-zero.

Figure 2.8 shows the structure of the filter. In the figure, the bits in the address bit-fields are hashed before using them to index into the arrays of counters.

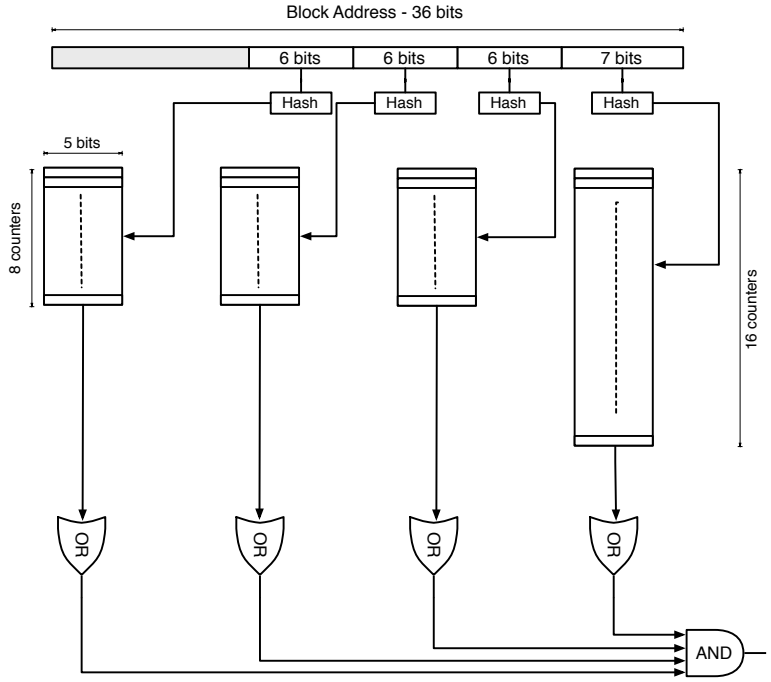


Figure 2.8: Per-bank Bloom filter in the proposed *Hierarchical* design. In the figure, H represents a hash operation.

2.4.3 Replacement of Entries in the Dedicated File

Since each Dedicated file only has a handful of MSHRs, it is easy to implement a FIFO replacement algorithm. Moreover, the contents of an MSHR are moved to the Shared file, they are never moved back to a Dedicated file. This general policy leverages the spatial locality of misses, to capture the active entries (those with frequent secondary misses) in the Dedicated files, and push the inactive entries (typically corresponding to long-latency main memory accesses) to the Shared file.

2.4.4 Complexity of the *Hierarchical* Implementation

Hierarchical is simple to implement. To start with, no changes are needed to the cache interface. Compared to other MHAs, any complexity of *Hierarchical* might come from four sources: allocating MSHRs, displacing them into the Shared file, handling replies from memory, and supporting the Bloom filter. Since the filter is a simple counter array that adds little complexity, the first three concerns are the focus of this section.

Before allocation, the MHA uses an *Available* signal to tell the cache bank that it has space to take in a new request. This signal is the logical OR of one signal coming from the Dedicated file in the bank and one from the Shared file. If neither file has space (*Available* is false), the cache bank locks-up. Otherwise, allocation proceeds in the Dedicated file. This step may involve a displacement.

The complexity of a displacement lies in solving three races or problems: (i) two Dedicated files want to displace into the same Shared file entry; (ii) an MSHR is needed while it is in transit from the Dedicated to the Shared file; and (iii) an entry being displaced finds that, despite initial indications to the contrary, there is no space in the Shared file. A simple algorithm avoids these problems. Specifically, on a displacement, the Dedicated file retains the MSHR being displaced and the corresponding bank of the L1 is locked-up to incoming requests, until the Shared file reports that it has taken in the data and the filter reports that it has been updated. Moreover, during the actual data transfer, the MSHR being displaced stays in the Dedicated file in locked state (inaccessible). If the transfer temporarily fails, the MSHR in the Dedicated file is unlocked, but the L1 bank remains locked-up until the whole process completes.

For replies coming from memory, the path just described is reused. Replies first check their home Dedicated file and, if they miss, they then check the Shared file. If a reply finds its corresponding MSHR in the locked state, it stalls until the MSHR is unlocked.

Overall, based on this discussion, I believe the *Hierarchical* has a simple implementation.

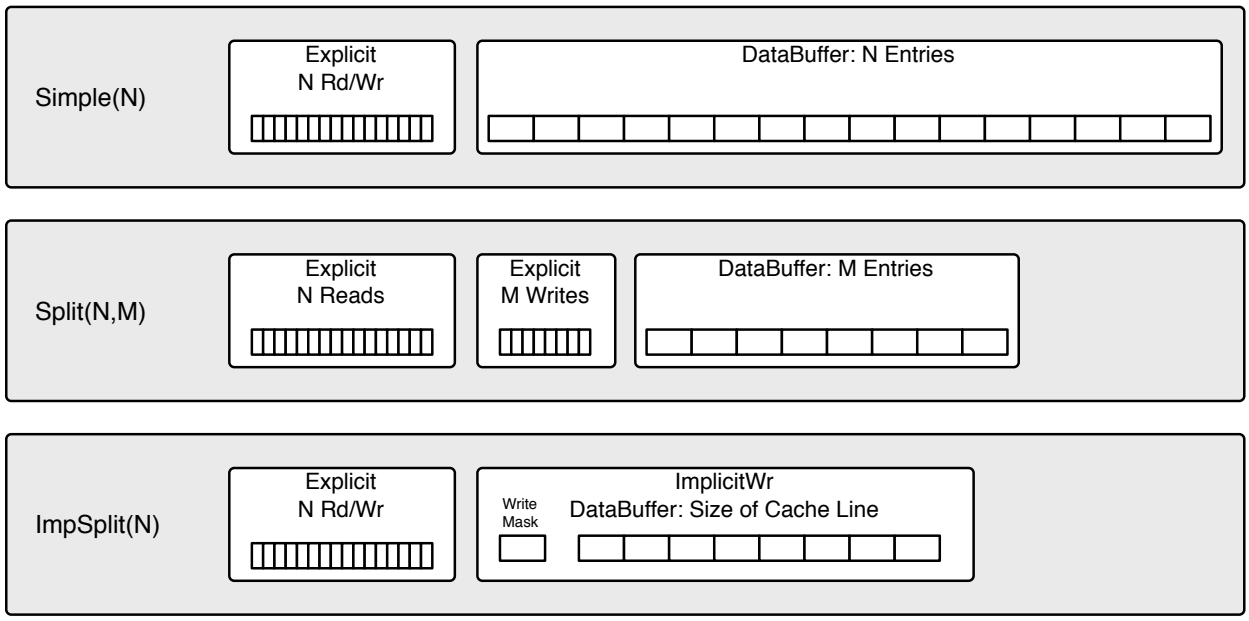


Figure 2.9: Three different MSHR organizations.

2.4.5 MSHR Organizations for High MLP

Three different MSHR organizations are considered. They extend Farkas and Jouppi's [28] Explicit and Implicit organizations. However, Farkas and Jouppi's MSHRs only record read information, since their caches are write-through and no-allocate. For the proposed designs, MSHRs also need to record write information. Kroft's design allocates an empty line in the cache immediately on a miss [42]. As a result, a write on a pending line deposits the update in the empty cache line. In the proposed case, cache misses can take a long time to complete. Therefore, allocating an empty line in the cache right away is undesirable. Instead, the MSHR organizations are designed with subentries that keep information on many read and write misses on the line.

The first of the three organizations is *Simple* (Figure 2.9). An MSHR includes an array of N explicit subentries, where each one can correspond to a read or a write. A read subentry stores the line offset and the destination register; a write subentry stores the line offset and a pointer into a companion N-entry data buffer where the update is stored. This organization leverages observations in Section 2.2.4 that MSHRs need to hold many read and/or write subentries. While this organization is simple, it has two shortcomings. First, to check for forwarding on a read, all the subentries in the MSHR need to be examined, which is time consuming. Second, this design consumes substantial area, since the data buffer needs to be very large in case all subentries are writes.

The *Split* organization separates N explicit read from M explicit write subentries (Figure 2.9). This design is motivated by the observation in Table 2.1 that many MSHRs do not need write subentries. Consequently, the number of supported write subentries are reduced to M, and only need M entries in the data buffer. This design improves area efficiency if M is significantly smaller than N. However, *Split* has the shortcomings of expensive checks for forwarding (like in *Simple*), and that it causes a stall if an MSHR receives more than M writes.

To solve these problems, *ImpSplit* keeps the explicit organization for N read subentries, but uses the implicit organization for writes (Figure 2.9). Each MSHR has a buffer for writes that is as large as a cache line, and a bit-vector mask to indicate which bytes have been written. Writes deposit the update at the correct offset in the buffer and set the corresponding bits in the mask. Multiple stores to the same address use a single buffer entry because they overwrite each other. Forwarding is greatly simplified because it only requires reading from the correct offset in the buffer. Moreover, this organization supports any number of writes at the cost of a buffer equivalent to a cache line; for the numbers of secondary write misses that observed in the experiments, this is area-efficient.

2.5 Experimental Setup

Execution-driven simulations are used to evaluate the MHA designs of Table 2.2 for the *Conventional*, *Checkpointed*, and *LargeWindow* processors. The architecture of the processors is shown in Table 2.5. *Conventional* is a 5-issue, 2-context SMT processor. *Checkpointed* extends *Conventional* with support for checkpoint-based value prediction like CAVA [11]. Its additional parameters in Table 2.5 are the Value Prediction Table and the maximum number of outstanding checkpoints. Each hardware thread has its own checkpoint and can rollback without affecting the other thread. *LargeWindow* is *Conventional* with a 512-entry instruction window and 2048-entry ROB.

The three processors have identical memory systems (Table 2.5), including two levels of on-chip caches. The exception is that *Checkpointed* has the few extensions required by the CAVA support [11], namely speculative state in L1 and a table with the value of the predictions made. All designs have a 16-stream strided hardware prefetcher that prefetches into L2 and, therefore, does not use L1 MSHRs.

The evaluation is performed using the cycle-accurate SESC simulator [65]. SESC models in detail the processor microarchitectures and the memory systems.

Area Design Point	MHA Design	MSHR Organization	Number of MSHRs	Tag & Data (Bytes)	Appx. Area (% L1)	Cycle Time (Cyc)	Acc. Time (Cyc)
Ordinary-Sized: 8% of L1 Area	Uni	ImpSplit(24)	8	1029	8	2	4
	Bank	ImpSplit(8)	2x8banks=16	1550	9	1	3
	Hier.	D: ImpSplit(4)	1x8banks=8	1227	8	1	3
		S: ImpSplit(24)	4			2	4
Medium-Sized: 15% of L1 Area	Uni	ImpSplit(32)	32	4620	15	2	4
	Bank	ImpSplit(8)	3x8banks=24	2325	15	1	3
	Hier.	D: ImpSplit(8)	2x8banks=16	2379	15	1	3
		S: ImpSplit(24)	8			2	4
Large-Sized: 25% of L1 Area	Uni	ImpSplit(32)	48	6930	24	2	4
	Uni 2p	ImpSplit(8)	8	773	25	2	4
	Bank	ImpSplit(12)	4x8banks=32	3352	26	1	3
	Hier.	D: ImpSplit(8)	2x8banks=16	5885	24	1	3
		S: ImpSplit(32)	30			2	4
Other MSHR Organizations at 15% of L1 Area	Bank	Simple(10)	3x8banks=24	2505	15	1	3
	Bank	Split(8,8)	3x8banks=24	2514	16	1	3
	Hier.	D: Simple(8)	2x8banks=16	2379	15	1	3
		S: ImpSplit(32)	8			2	4
	Hier.	D: Split(8,8)	2x8banks=16	2065	16	1	3
		S: ImpSplit(24)	8			2	4

Table 2.4: Area, cycle time, and access time for the MHA designs and MSHR organizations considered.

2.5.1 Comparing MHAs That Use the Same Area

MHAs offer a large design space from which a designer must choose. In this work, different designs that use the *same area* are compared. This is a fair constraint given that structure has a large impact on capacity at a given area. Three design points are considered: a *Medium-Sized* design, where the MHA uses an area equivalent to 15% of the area of the 8-bank 32-Kbyte L1 cache; a *Large-Sized* design, where it uses the equivalent of 25% of the cache area; and an *Ordinary-Sized* design where, like the Pentium 4 MSHR file, the MHA uses the equivalent of ≈8% of the cache area.

To estimate area, the newly available CACTI 4.1 [82] is used. This version of CACTI is more accurate than the previous 3.2 version, as it has been specifically designed for nanoscale technologies such as the 65nm one considered here. See [86] for details on the CACTI runs. As of September 2006, the authors of CACTI acknowledge a bug in the area calculation for a banked cache. For all the experiments, CACTI 4.1 is modified to correct the bug.

Combinations of MHA design, MSHR organization, number of MSHRs, number of subentries, and associativity are considered that match each of the three target area points, or come very close. An automated script generates all possible combinations, and computes area, cycle time, and access time. A cycle-accurate processor-memory simulator is used to evaluate the overall performance of workloads using each design.

Table 2.4 lists the best designs found. Column 1 shows the four sets of experiments that

<table>
<tr><th>All</th><th>Memory System</th></tr>
<tr><td>Frequency: 5.0 GHz at 65nm
Fetch/issue/comm width: 6/5/5
LdSt/Int/FP units: 4/3/3
SMT contexts: 2
Branch penalty: 13 cyc (min)
RAS: 32 entries
BTB: 2K entries, 2-way assoc.
Branch predictor (spec. update):
bimodal size: 16K entries
gshare-11 size: 16K entries</td><td>
<table>
<tr><th></th><th>I-L1</th><th>D-L1</th><th>L2</th></tr>
<tr><td>Size:</td><td>32KB</td><td>32KB</td><td>2MB</td></tr>
<tr><td>Assoc:</td><td>2-way</td><td>2-way</td><td>8-way</td></tr>
<tr><td>Line size:</td><td>64</td><td>64</td><td>64</td></tr>
<tr><td>RT:</td><td>2 cyc</td><td>3 cyc</td><td>15 cyc</td></tr>
<tr><td>Ports/Bank:</td><td>2</td><td>1</td><td>1</td></tr>
<tr><td>Banks:</td><td>–</td><td>8</td><td>–</td></tr>
</table>
HW Pref.: 16-stream strided
(bet. L2 and mem.)
Mem Bus Bandwidth: 15GB/s
Mem RT: 500 cyc
</td></tr>
<tr><td>Conventional and Checkpointed</td><td>LargeWindow</td></tr>
<tr><td>I-window/ROB size: 92/192
Int/FP registers: 192/192
Ld/St queue entries: 60/50

Checkpointed Only:
Val. Pred. Table: 2048 entries
Max Outs. Ckps: 1 per context</td><td>I-window/ROB size: 512/2048
Int/FP registers: 2048/2048
Ld/St queue entries: 768/768</td></tr>
</table>

Table 2.5: Processors simulated for MHA study. In the table, RAS and RT stand for Return Address Stack and minimum Round-Trip latency from the processor, respectively. Cycle counts refer to processor cycles.

were performed. The top three compare MHA designs that take the equivalent of 8%, 15%, and 25% of the L1 area, respectively. For each experiment, the best *Unified*, *Banked*, and *Hierarchical* designs are used — although, for the 25% area experiment, two different *Unified* designs are considered, as discussed later. The third column shows the MSHR organization used, with the number of explicit subentries in parenthesis as in Figure 2.9. All the best designs use the *ImpSplit* MSHR organization. For completeness, a fourth experiment (last row of Column 1 of Table 2.4) is performed comparing designs that use *Simple* and *Split* organizations.

For each MHA design and MSHR organization, Table 2.4 shows the number of MSHRs used (Column 4), their associativity (Column 5), the size of the tag and data arrays in bytes (Column 6), the area of the tag and data arrays as a fraction of the L1 area (Column 7), the cycle time (Column 8), and the access time (Column 9). All cycle counts are in processor cycles. The size and area of *Hierarchical* are the addition of the contributions of the Dedicated and Shared files. All of these structures are pipelined except the Shared file in *Hierarchical*.

The L1 is a 32 Kbyte 2-way cache organized in 8 banks with 1 read/write port per bank. Such a cache in 65nm technology is estimated to be 0.6965 mm^2. It is simulated it with a

SPECint2000	SPECfp2000
256.bzip2 (*bzip2*)	188.ammp (*ammp*)
254.gap (*gap*)	173.applu (*applu*)
181.mcf (*mcf*)	179.art (*art*)
253.perlbmk (*perlbmk*)	183.equake (*equake*)
	177.mesa (*mesa*)
	172.mgrid (*mgrid*)
	171.swim (*swim*)
	168.wupwise (*wupwise*)
Mix	
179.art, 183.equake (*artequake*)	
179.art, 254.gap (*artgap*)	
179.art, 253.perlbmk (*artperlbmk*)	
183.equake, 253.perlbmk (*equakeperlbmk*)	
177.mesa, 179.art (*mesaart*)	
172.mgrid, 181.mcf (*mgridmcf*)	
171.swim, 181.mcf (*swimmcf*)	
168.wupwise, 253.perlbmk (*wupwiseperlbmk*)	

Table 2.6: Workloads used in the experiments.

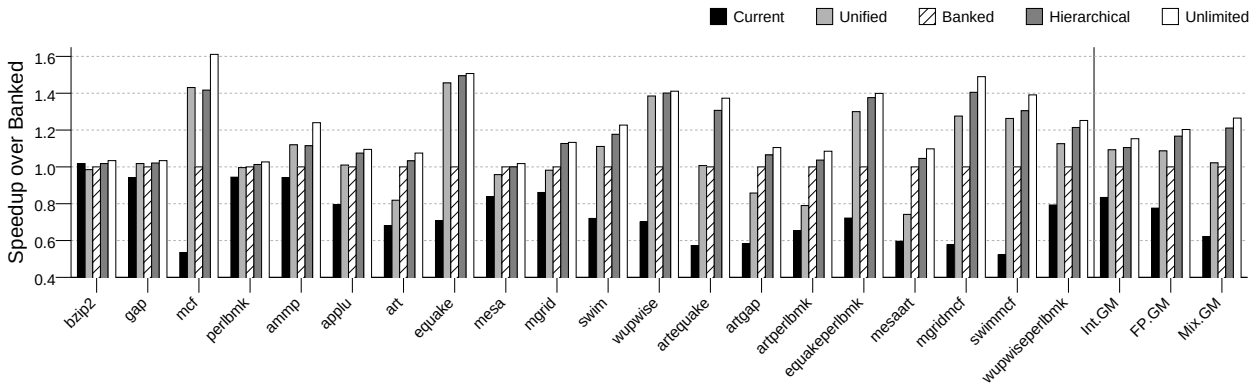


Figure 2.10: Performance of the different MHA designs at the 15% target area for the *Checkpointed* processor.

cycle time of 1 cycle and an access time of 3 cycles.

2.5.2 Workloads

Experiments were conducted using SPECint2000 codes, SPECfp2000 codes, and workload mixes that combine two applications at a time (Table 2.6). The infrastructure does not support *eon*, which is written in C++. Moreover, there are 7 SPECint codes that have so few misses that a *perfect* MHA (unlimited number of MSHRs, subentries, and bandwidth) makes not even a 5% performance impact in any of the architectures analyzed. Consequently, only

the remaining 4 SPECint codes are analyzed in the rest of the chapter. In the summary section, the performance impact is averaged out for all designs over the 11 SPECint applications that are simulated.

From SPECfp, all the applications were used except for six that the infrastructure does not support (four that are in Fortran 90 and two that have unsupported system calls). Finally, for the workload mixes, the algorithm followed is to pair one SPECint and one SPECfp such that one has high MSHR needs and the other low. In addition, one mix combines two lows, two combine two highs, and two others combine 2 SPECfps. Overall, a range of behaviors is covered in the mixes. In these runs, each application is assigned to one hardware thread and the two applications run concurrently.

The codes are compiled using gcc 3.4 -O3 into MIPS binaries and use the *ref* data set. Each program is evaluated for 0.6-1.0 billion committed instructions, after skipping several billion instructions as initialization. Performance is compared using committed IPC. When comparing the performance of multiprogramming mixes, weighted speedups are used as in [88].

2.6 Evaluation

In this evaluation, the performance of the different MHA designs at the Medium-Sized area point (15% of the L1 area) and at other area points, characterize *Hierarchical*, and evaluate different MSHR organizations. Unless otherwise indicated, the MHA designs are those shown in the first three rows of Column 1 of Table 2.4. Also, plots are normalized to the performance of *Banked*.

2.6.1 Performance of MHA Designs at 15% Area

Figure 2.10 compares the performance of the different MHA designs of Table 2.4 at the 15% target area for the *Checkpointed* processor. As a reference, the figure also includes *Current* and *Unlimited*. The former is a design like that of Pentium 4 (Table 2.2); the latter is an infeasible MHA design that supports an unlimited number of outstanding misses with unlimited bandwidth. The rightmost three sets of bars in the figure show the geometric mean of the integer, FP, and mix workloads.

The *Current* design is much worse than the other MHAs for *Checkpointed* processors. Such processors are bottlenecked by *Current* and need aggressive MHA designs. For example, *Hierarchical* speeds-up execution over *Current* by a geometric mean of 32% for SPECint, 50% for SPECfp, and 95% for mixes. These are substantial speedups.

Among the aggressive designs, *Hierarchical* performs the best, and is very close to *Unlimited*. Compared to *Unified*, *Hierarchical* has lower capacity but, thanks to the three techniques of Table 2.3, it offers higher bandwidth to accesses. As a result, *Hierarchical* is faster than *Unified* by a geometric mean of 1% (SPECint), 7% (SPECfp), and 18% (mixes). The speedups are highest for high-MLP scenarios, such as when these SMT processors run a multiprogrammed load.

On average, *Banked* is worse than *Unified*. This is because the higher bandwidth that it provides is not fully leveraged due to access imbalance (section 2.2.3). There are some exceptions, such as the workloads with *art*, which benefit more from higher bandwidth than are hurt by imbalance. On average, *Hierarchical* is faster than *Banked* for the three workload groups by a geometric mean of 10%, 16%, and 21%. Overall, *Hierarchical* delivers significant improvements over the other aggressive designs for a very modest complexity (Section 2.4.4).

For completeness, the effect of MHA designs on *Conventional* and *LargeWindow* processors is examined, although only the geometric mean is shown. They are shown in Figures 2.11(a) and 2.11(b), respectively. With *Conventional* processors, the performance difference between *Current* and the aggressive designs is much smaller. This shows that state-of-the-art, relatively low-MLP processors cannot leverage aggressive MHAs as much. Still, *Hierarchical* and *Banked* are consistently the best.

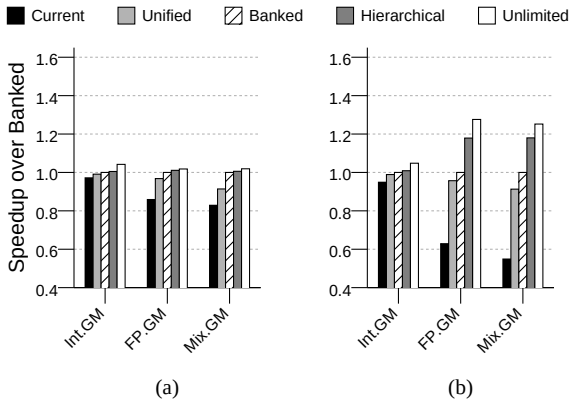
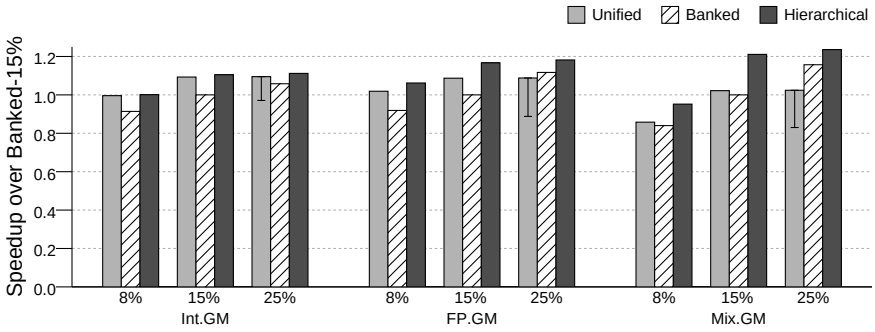
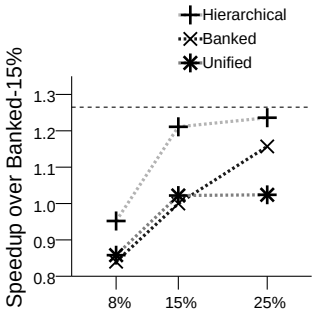


Figure 2.11: Performance of the different MHA designs at the 15% target area for the (a) *Conventional* processor and (b) *LargeWindow* processor.

With *LargeWindow* processors (Figure 2.11(b)), *Current* bottlenecks the processor, and that *Hierarchical* is significantly faster than the other two aggressive MHA designs — *Hierarchical* is faster than *Unified* by 2%, 23%, and 29% in the three workload groups. In this processor, the profile of speedups is somewhat different than in *Checkpointed* — most notably, *Banked* is better than *Unified*, and SPECint codes show smaller speedups. The



(a)



(b)

Figure 2.12: Performance of the *Checkpointed* processor for different MHA designs, and target areas 8%, 15% and 25%. Chart (a) shows the average for each set of applications. The line segments in the bars for 25% *Unified* indicate the reduction in performance if we use the dual-ported *Unified* design. Chart (b) repeats the information in *Mix.GM* of Chart (a) in a different format. The dashed horizontal line shows the performance of *Unlimited*.

reason for the first effect is that *LargeWindow*'s outstanding misses require more bandwidth than the *Checkpointed* ones (Figure 2.4). Consequently, *Banked* works relatively better. The reason for the second effect is that, thanks to value prediction and speculative retirement, *Checkpointed* presents a longer effective window than *LargeWindow* for SPECint codes, which enables higher speedups for these codes. Overall, these results show that processors other than *Checkpointed* can also use aggressive MHA designs. Most likely, any high-MLP architecture will benefit from aggressive MHA designs.

2.6.2 Performance at Different Area Points

Hierarchical maintains its performance advantage over *Banked* and *Unified* across a wide range of area points. This is seen for *Checkpointed* in Figure 2.12. Figure 2.12(a) is organized per workload type. For each workload, going from left to right, the target area increases from

8% (Ordinary-Sized design) to 15% (Medium-Sized design) and 25% (Large-Sized design). In each workload, the bars are normalized to *Banked* with 15%.

At each workload and area point, *Hierarchical* is the fastest design. Moreover, *Hierarchical* at the 15% target area is better than *Unified* or *Banked* at the 25% target area — which use much more area.

Unified is most competitive at the 8% target area, where its better use of area relative to the other designs has the highest impact. As the target area scales up, however, the performance of *Unified* levels out, even though its capacity is the highest (Table 2.4). The lower bandwidth of *Unified* prevents it from exploiting its higher capacity. To address this problem, also evaluated is a second design for *Unified* at the 25% target area: one with two ports (Table 2.4). This *Unified* design has higher bandwidth but, to keep the area constant, its number of MSHRs is reduced significantly to 8. The performance of this design is shown in Figure 2.12(a) as the lower end of the line segments in the *Unified* 25% bars. Performance is always lower than the *Unified* 25% single-ported design. Even though the dual-ported design has higher bandwidth, it is crippled by its low capacity.

Banked is unattractive for the 8% and 15% target areas. This is because it suffers from load imbalance due to its low per-bank capacity. As it gains capacity at the 25% target area, it outperforms *Unified* for *FP* and *Mix* workloads, although it does not match *Hierarchical* yet.

Figure 2.12(b) repeats the information in *Mix.GM* of Figure 2.12(a) in a format that emphasizes the scaling trends of each design. In the figure, the performance of *Unlimited* is shown as the dashed horizontal bar. *Hierarchical* offers the highest performance across all area points, obtaining close to *Unlimited* performance already at 15% area. *Unified*'s performance saturates at around the 15% target area due to limited bandwidth. Only by adding a second port at the cost of much higher area can *Unified* achieve better performance. Finally, *Banked* improves its performance as the target area increases. Eventually, there may be a point where it will match the performance of *Hierarchical*. However, such a design point will be much less area-efficient than the ones presented here.

2.6.3 Characterization of *Hierarchical* at 15% Area

Table 2.7 characterizes *Hierarchical* for *Checkpointed* at the 15% target area. The first group of columns (Columns 2-4) shows how the L1 misses are processed by the different files in *Hierarchical*. Misses can be of three types: primary (Column 2), secondary that hit in a Dedicated file (Column 3), and secondary that hit in the Shared file (Column 4). Primary misses create an entry in a Dedicated file. Of the three types of misses, only the last one

Workload	L1 Miss Breakdown			Accesses Removed (%)	Displacement Stats	
	Primary (%)	Dedicated Hit(%)	Shared Hit(%)		Sub Full (%)	L2 Miss (%)
bzip2	57.3	28.6	14.0	76.1	12.1	0.3
gap	17.9	39.1	43.0	99.0	80.1	47.2
mcf	39.2	37.9	22.8	95.4	27.3	84.1
perlbmk	34.2	58.7	7.2	96.6	43.0	35.9
ammp	40.5	55.5	4.1	56.1	29.7	52.3
applu	28.1	55.4	16.5	87.6	68.7	20.7
art	79.4	19.0	1.6	98.6	0.1	18.3
equake	26.3	30.6	43.1	96.3	59.2	69.2
mesa	35.2	51.5	13.3	97.6	33.9	21.2
mgrid	44.3	30.1	25.6	89.9	2.3	7.1
swim	39.0	32.1	28.9	99.8	24.7	49.9
wupwise	12.0	35.0	53.0	97.9	76.4	74.8
artequake	63.7	22.2	14.2	97.6	7.3	25.9
artgap	65.3	26.8	7.9	98.9	1.6	24.1
artperlbmk	72.8	22.4	4.8	97.9	1.1	17.3
equakeperlbmk	29.7	37.1	33.2	96.2	51.0	56.7
mesaart	67.7	25.0	7.3	98.1	2.5	15.6
mgridmcf	53.6	25.9	20.5	83.0	3.9	19.6
swimmcf	44.1	31.5	24.4	98.8	17.8	74.4
wupwiseperlbmk	26.4	47.9	25.7	98.0	61.8	28.4
Int.Avg	37.2	41.1	21.8	92.0	40.6	41.9
FP.Avg	38.1	38.6	23.3	89.9	36.9	39.2
Mix.Avg	52.9	29.8	17.3	95.9	18.4	32.7

Table 2.7: Characterization of the dynamic behavior of *Hierarchical* for *Checkpointed* at the 15% target area.

involves storing information on the miss in the Shared file. Overall, this happens for only 17-23% of the L1 misses on average.

Column 5 shows the effectiveness of the Bloom filter at saving accesses to the Shared file. The numbers listed are the fraction of primary misses that are prevented from accessing the Shared file by the Bloom filter — in other words, the fraction of *unnecessary* accesses to the Shared file that are eliminated by the Bloom filter. They are unnecessary because they would miss in the Shared file anyway. The numbers shown are averages across all banks. From the table, it is shown that the Bloom filter eliminates on average 90-96% of useless accesses to the Shared file. It does not remove them all because of false positives. Overall, the Bloom filter ensures that the Shared file does not become a bottleneck.

The third group of columns (Columns 6-7) shows data regarding the displacement of entries from the Dedicated files to the Shared file. Column 6 shows the fraction of such displacements triggered by lack of subentries. This number is on average 18-41%. The other displacements are triggered by lack of entries. Column 7 shows the fraction of all the dis-

Workload	L1 Miss Rate (%)	L2 Miss Rate (%)	MSHR Fwd L1 misses (%)	Bus Util (%)
bzip2	3.8	0.0	1.0	0.1
gap	0.8	0.2	49.1	4.4
mcf	15.9	7.6	4.6	45.8
perlbmk	3.3	0.2	6.6	2.7
ammp	18.4	1.7	0.8	4.5
applu	3.0	0.2	12.7	7.0
art	44.2	4.0	0.0	27.1
equake	5.3	1.7	2.0	23.3
mesa	4.8	0.2	7.3	4.8
mgrid	15.0	1.1	0.5	20.2
swim	7.3	2.8	0.7	45.1
wupwise	1.6	1.0	4.8	15.6
artequake	23.4	3.0	0.7	31.5
artgap	27.6	3.2	1.0	31.6
artperlbmk	31.3	2.6	0.4	24.5
equakeperlbmk	6.2	1.4	3.0	23.1
mesaart	28.2	2.2	0.8	25.1
mgridmcf	20.4	3.2	0.9	46.1
swimmcf	11.3	4.7	2.1	51.2
wupwiseperlbmk	4.9	0.3	8.6	8.7
Int.Avg	6.0	2.0	15.3	13.3
FP.Avg	12.5	1.6	3.6	18.5
Mix.Avg	19.2	2.6	2.2	30.2

Table 2.8: Continued characterization of the dynamic behavior of *Hierarchical* for *Checkpointed* at the 15% target area.

placements that are also L2 cache misses. Such fraction is on average 33-42%. Consequently, the Shared file often holds long-latency misses.

Figure 2.8 shows information about the rest of the memory system: L1 and L2 miss rates, fraction of L1 read misses that get their data forwarded from an MSHR (a significant 15% for SPECints), and bus utilization.

Finally, to assess the frequency of cache lock-up, Figure 2.13 shows the fraction of the time during which *at least one* of the cache banks is locked-up — because of lack of either entries (MSHRs) or read subentries (not due to write subentries because *ImpSplit* MSHRs are used). For each workload, the bars are normalized to *Banked*.

The figure shows that the fraction of time with lock-up tends to increase going from *Unified* to *Hierarchical* and to *Banked*. As discussed in Section 2.2.3, the reason is that load imbalance in banked MHA designs causes some banks to fill up sooner. However, while *Unified* typically has the least lock-up time, it does not have the highest performance; it is hurt by lower bandwidth. The figure also shows that most of the lock-up is due to

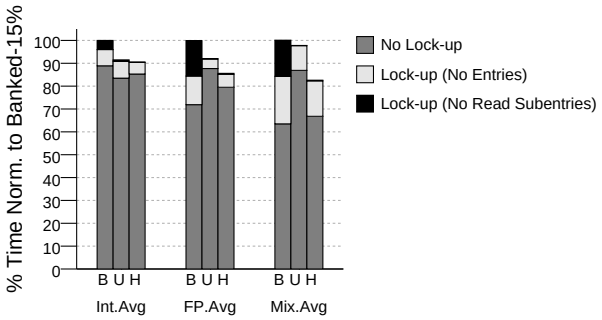


Figure 2.13: Breakdown of the execution time for *Banked* (B), *Unified* (U), and *Hierarchical* (H) at the 15% target area.

lack of entries rather than subentries. Overall, *Hierarchical* performs the best because it accomplishes both high bandwidth and modest lock-up time.

2.6.4 Evaluation of Different MSHR Organizations

Now, the effect of the different MSHR organizations of Figure 2.9 in *Banked* and *Hierarchical* for *Checkpointed* at the 15% target area is compared. The *ImpSplit*-based MHA designs used so far are compared to the *Simple*- and *Split*-based MHA designs of the last row of Column 1 of Table 2.4. Figure 2.14 shows the performance of the different MHA designs normalized to *Banked-ImpSplit*.

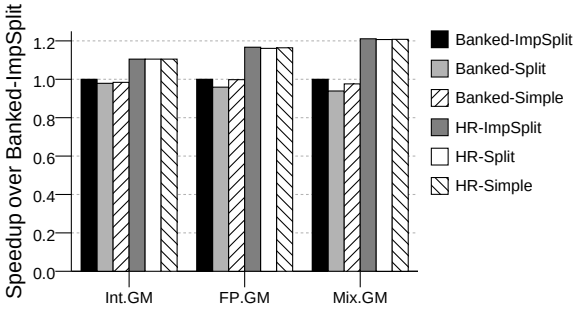


Figure 2.14: Performance of the *Checkpointed* processor with different MSHR organizations. The target area is 15%. In the figure, *HR* stands for *Hierarchical*.

In *Banked*, *Split* performs slightly worse than *ImpSplit* due to a lack of write subentries. *Simple* performs roughly as well as *ImpSplit*; even though it has a few more read subentries, it does not have as much capacity for write misses. In *Hierarchical*, the changes only have a small impact. Part of the reason is that the Shared file minimizes the differences. Overall, the *ImpSplit* organization is selected because it performs as well or better than the other

organizations and supports a simpler implementation of read forwarding (Section 2.4.5).

2.7 Summary

Recently-proposed processor microarchitectures that require substantially higher MLP promise major performance gains. Unfortunately, the MHAs of current high-end systems are not designed to support the necessary level of MLP. This chapter focused on designing a high-performance, area-efficient MHA for these high-MLP microarchitectures.

This work made three contributions. First, it showed that state-of-the-art MHAs for L1 data caches are unable to leverage the new microarchitectures. Second, it proposed a novel, scalable MHA design that supports these microarchitectures. The proposal's key ideas are: (i) a *hierarchical* organization for high bandwidth and minimal cache lock-up time at a reasonable area cost, and (ii) a *Bloom filter* that eliminates most of the unneeded accesses to the large MSHR file.

The third contribution was the evaluation of an MHA design in a high-MLP processor. This study focused mostly on a design point where the MHA uses an area equivalent to 15% of that of the L1 data cache. Compared to a state-of-the-art MHA, *Hierarchical* delivers geometric-mean speed-ups of 32% for a subset of SPECint (or a geometric mean of 11% for the 11 SPECint applications that were simulated, including those with very few misses), 50% for SPECfp, and 95% for multiprogrammed loads. The proposed design was also compared to two extrapolations of current MHA designs, namely *Unified* and *Banked*. For the same area, *Hierarchical* speeds-up the workloads by a geometric mean of 1-18% over *Unified* and 10-21% over *Banked* — all for a very modest complexity. Finally, the proposed design performed very close to an unlimited-size, ideal MHA.

Chapter 3

CAP: Criticality Analysis for Power-Efficient Speculative Multithreading

Relentless transistor integration is driving processor manufacturers to build Chip Multiprocessor (CMP) architectures. However, while CMPs can effectively speed-up parallel programs, much of the application base today is still composed of sequential applications — for example, non-numerical applications that compilers fail to parallelize.

A proposed solution to speed-up these hard-to-parallelize codes is Speculative Multithreading (SM) (e.g., [34, 41, 48, 76, 79, 84, 83]). In SM, sequential applications are partitioned into tasks, which are then speculatively executed in parallel, hoping not to violate the sequential semantics of the program. As tasks execute, special hardware monitors their data accesses and checks for cross-task dependence violations at run-time. If any violation detected, the hardware squashes the offending task(s), repairs the program state, and restarts them.

While evaluations of SM on a CMP have generally shown good, if modest, speedups, an important concern has been the power inefficiency of aggressive speculation. Indeed, as more tasks are executed speculatively to deliver higher speedups, there is a higher chance of spending power on work that ultimately gets squashed. Wasting power is a very unattractive proposition, as power and energy consumption are currently major constraints in processor design.

Given the key importance of power issues, the goal is to design power-efficient SM systems. Previous work on this area by Renau *et al.* [66] focused on improving the energy efficiency of SM operations. In this chapter, the interaction between the tasks of an application is the main focus instead. Specifically, I make a key observation on the behavior of SM tasks: not all of the tasks that are running in an SM environment are equally critical for performance and power-efficient execution of the application — some are more critical than others.

To leverage this insight, two architectural features are needed. First, the CMP has to be able to assess the criticality of each task. Previous work on criticality analysis focused on instruction-level criticality [30, 43, 68, 89, 90]. While some of the ideas can be reused for tasks, the model and hardware implementation required are substantially different. Second, the CMP must be able to schedule tasks in a power efficient way in accordance with their

criticality. This can be supported with Dynamic Voltage and Frequency Scaling (DVFS) on a per-core basis.

Based on these observations and needs, I propose *CAP*, a novel architecture that (i) analyzes and predicts the criticality of tasks in a SM application at run-time, and (ii) uses criticality to schedule tasks on a SM CMP with per-core DVFS for power-efficient execution. Critical tasks are scheduled on fast cores, while non-critical ones run on slower ones.

Overall, this work makes three contributions:

1. It develops a widely-applicable, novel task-criticality model for SM. The model is stored in hardware in a special on-chip module and is refined as execution proceeds.

2. It designs the CAP architecture. Experiments with SPECint, SPECfp and Olden applications show that CAP reduces the average dynamic power of an optimized baseline by 24%, 35%, and 34% respectively, while degrading performance by only 6%, 7%, and 5% on average. Furthermore, it reduces $E \times D^2$ by 9%, 21%, and 23% on average respectively. Compared to scheduling based on task ordering, CAP can reduce $E \times D^2$ by as much as 61%.

3. It characterizes the task criticality composition of different applications.

The rest of the chapter is organized as follows. Section 3.1.2 presents a background; Sections 3.2 and 3.3 present the proposed criticality model and the CAP architecture, respectively; Sections 3.4 and 3.5 evaluate CAP; and Section 3.6 summarizes.

3.1 Background

In this section, prior work in the areas of critical-path analysis and speculative multithreading is explored.

3.1.1 Instruction-Level Criticality Analysis

The critical path is the chain of dependent events that determine the overall execution time of the program. Delaying events in the critical path hurts performance more than delaying other events. Knowing the criticality of each event can help processors allocate resources more efficiently and eliminate useless work.

There has been substantial research on predicting the critical instructions and quantifying instruction criticality [30, 43, 68, 89, 90]. Dependence graph-based schemes [30, 43], like the one proposed in the this work, generate a dynamic dependence graph to estimate the

critical path during or after program execution. Each node represents a pipeline stage, such as dispatch, execute or retire. Directed edges between nodes represent the dependence relation, such as control dependence, data dependence or resource dependence. Critical path information has been used to implement cost-sensitive policies for several uses, such as scheduling and steering instructions on clustered microarchitectures [30, 68, 89]; reducing power by sending critical instructions to fast (high power) functional units and non-critical ones to slow (low power) functional units [9, 29, 71]; and controlling misprediction by only predicting for critical instructions [30, 56, 89].

The instruction-level critical path analysis has the advantage of reasoning about events that happened within a microprocessor, such as contention on functional units. However, in the scenario of multithreaded architectures like SM, directly applying the instruction-level model will cause many problems. First, the flood of per-instruction information makes the model hard or impossible to implement in hardware. Second, most per-instruction information is useless when considering interactions between tasks in multithreaded architectures. For example, instructions that are not responsible for inter-thread communication are usually uninteresting. Third, the task-level information is scattered into instructions belonging to that task. The conglomeration of instruction history into task-level information is not trivial to implement in practice.

Due to the limitations of instruction-level criticality analysis for SM, in this work, a task-level criticality model is proposed which tracks and collects *per-task* information, and then predicts the criticality of each task. The number of tasks is orders of magnitude lower than the number of instructions. Therefore, it is possible to store and analyze the model in hardware. Moreover, the model can easily focus on the per-task information, thereby avoiding any aggregation of instructions.

3.1.2 Speculative Multithreading (SM)

SM extracts tasks from a sequential program and executes them in parallel, hoping not to violate sequential semantics. The control flow of the sequential program imposes a task order. Therefore, the term predecessor and successor tasks are used. The safe (or non-speculative) task precedes all speculative tasks. SM schemes provide special hardware support to detect if the parallel execution of speculative tasks violates any data dependence relation required by the sequential program. If any dependence is violated, the offending tasks are squashed, any polluted state is repaired and the tasks are re-executed.

Multi-Versioned Caches. Under SM, speculative state generated by speculative tasks has to be stored separately before it can be merged with the safe state of the program.

Speculative data can be saved in a special hardware buffer or the cache of the processor. A cache that can hold state from multiple speculative tasks is called *multi-versioned*. Multi-versioning allows multiple speculative tasks to run on a single processor. It also enables a few performance optimizations, such as: avoiding processor stall when tasks are heavily load-imbalanced, enabling lazy commit [62], and enabling speculative task preemption. With speculative task preemption, a running speculative task is temporarily suspended and the core is given to another task. Speculative task preemption is used in this work.

In-Order and Out-of-Order Task Spawning. There are two types of task spawning schemes in SM: in-order and out-of-order [67]. Under in-order spawning, an individual task can at most spawn one correct task in its lifetime. A correct task is one that is in the sequential execution of the program, rather than in the wrong path of a branch. As a result, correct tasks are spawned in-order, namely, in the same order as in sequential execution. Under out-of-order spawning, an individual task can spawn multiple correct tasks. Out-of-order spawning is harder to support, but it enables more task parallelism: two code sections that are far-off in sequential execution can be executing in parallel before some of their intervening code sections have even been spawned. The criticality model is general enough to support *both* in-order and out-of-order task spawning.

Power-efficient Speculative Multithreading. Not much work has been done in the area of power-efficient SM. Renau *et al.*'s paper [66] is the first to analyze the energy efficiency of a SM system. They identified and quantified the main sources of energy consumption in SM. Then a set of simple energy-saving optimizations was proposed for SM. Their work focuses on improving the efficiency of SM operations, however, this work focuses on the interaction between SM tasks.

Nagpal and Bhowmik [56] also used an instruction-level criticality model to drive energy-aware speculation for SM processors. They focused on controlling misspeculation by delaying the non-critical loads and enhancing the branch prediction. This scheme differs in that it exploits the criticality of task interactions, not just misspeculation, to improve power efficiency.

3.2 Task-Level Criticality Model

This section describes a novel task-level criticality model for SM environments. The model is very general, as it applies to both SMT- and CMP-based SM architectures, both with in-order and out-of-order task spawning. Since it tracks only task-level information, it significantly reduces storage requirements compared to an instruction-level scheme, and lends itself to a

simpler hardware implementation.

3.2.1 Lifetime of a SM Task

In an instruction-level criticality model, an instruction is represented by a set of nodes that represent different stages in the instruction's lifetime — e.g., Dispatch (or Fetch), Execute and Retire. Similarly, a task experiences a few stages during its lifetime: Start (the task is created by a predecessor), Execute (the task is assigned to a free core or context), Finish (the execution reaches the end instruction) and Commit (the commit token for the task is consumed). Figure 3.1(a) shows a task's lifetime, where the thicker line represents the use of a core or context.

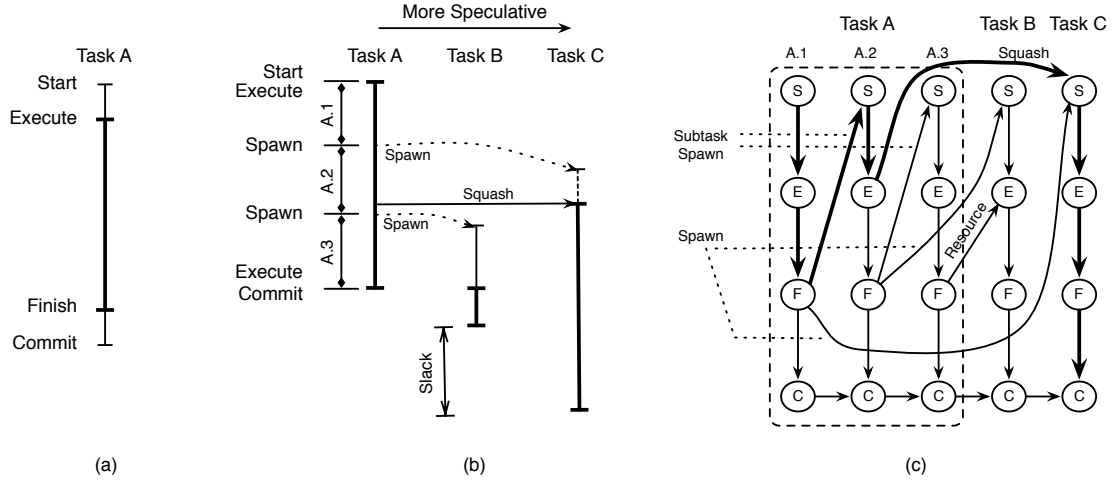


Figure 3.1: An example of the task-level criticality model. Shown above are the stages in a task lifetime (a); example of task execution where Task A spawns C, then squashes it, and finally spawns a less-speculative Task B (b); and resulting criticality graph (c).

To accurately calculate the critical path in a set of dynamic SM tasks, the model must capture all the interactions between tasks. Specifically, a task can execute a spawn instruction, thereby creating another task. Moreover, a task can execute a store instruction that squashes a successor, forcing the latter and its successors to re-start or synchronize. In addition, a task may have to wait for another to finish execution and release the core or context. Finally, a task may have to wait for another to commit, so that it can become safe and either commit or perform an irreversible operation such as I/O.

In addition, a task lifetime is divided into stages between which these interactions take place. Note that using the four obvious stages Start, Execute, Finish and Commit as in Figure 3.1(a) is suboptimal. The reason is that the Execute stage can contain instructions

with a *mix* of levels of criticality. Specifically, if it contains a task spawn instruction, the instructions before the spawn may be much more or less critical than the ones after it. Moreover, if the stage includes a write that squashes or synchronizes with a successor task, the instructions before and after the write may have different criticality levels.

3.2.2 Proposed Task-Level Criticality Model

In the model, the challenge of supporting spawn, squash, and synchronization on the Execute stage are each handled differently. Consider spawns first. In an environment with in-order spawning, since a task can only spawn once, the Execute stage can be divided into two substages: one before and one after the spawn. Unfortunately, in the case of out-of-order spawning, a task may spawn multiple tasks. Dividing the task into more stages would make the model difficult to use, as instances of a given task could look very differently.

To solve this problem, the model compensates by breaking a task into two *subtasks* at every spawn instruction. As an example, consider Figure 3.1(b). It shows Task A spawning Task C and then squashing it. After C restarts, Task A spawns a less speculative Task B, which cannot get a processor and waits until Task A finishes and relinquishes the processor. After B completes, C is still running. In this example, there are three subtasks in A: from the start of the task to the first spawn (subtask *A.1* in Figure 3.1(b)); between two spawns (subtask *A.2*); and from the last spawn to the commit of the task (subtask *A.3*).

This approach, where a subtask has at most one spawn, has two important properties. First, in-order and out-of-order task spawning environments are handled seamlessly. Secondly, a task spawn cannot cause instructions with different criticality levels to be included in the same Execute node.

This approach is harder to apply to writes that cause squash or synchronization operations. The reason is that the task that issues the write does not know, at the time of the write, that it will cause a squash or sync — only later, when the invalidation propagates to other caches is it declared. For this reason, and because squashes and syncs are rarer than spawns, the Execute nodes are not broken when a write causes a dependence event. This simplification degrades the effectiveness of the model only slightly, since the Execute node may contain instructions from different criticality levels.

Overall, a subtask is modeled with the four nodes shown in Table 3.1. Start (S) and Execute (E) follow the description of Section 3.2.1. Finish (F) corresponds to the point after the subtask executes a spawn or a task-end instruction. Finally, Commit (C) involves receiving the commit token, committing the architectural state of the whole task (if this is the last subtask in the task), and passing the token to the next subtask.

Node	Description
Start (S)	Subtask is created
Execute (E)	Subtask executes
Finish (F)	Subtask completes a spawn or a task-end instruction
Commit (C)	Subtask commits whole task state (if this is the last subtask in the task) and passes token

Table 3.1: Nodes per subtask in the criticality graph.

Edge	Description	
$S_i \rightarrow E_i$ $E_i \rightarrow F_i$ $F_i \rightarrow C_i$	Transitional edges within a subtask	
$F_i \rightarrow S_j$	*Spawn* edge	Subtask(i) spawns Subtask(j)
$E_i \rightarrow E_j$	*Sync* edge	Subtask(i) syncs with Subtask(j)
$E_i \rightarrow S_j$	*Squash* edge	Subtask(i) squashes Subtask(j)
$F_i \rightarrow E_j$	*Resource* edge	Subtask(i) relinquishes core to Subtask(j)
$C_i \rightarrow E_{i+1}$	*BeSafe* edge	Subtask(i+1) must wait to be safe
$C_i \rightarrow C_{i+1}$	*Commit* edge	Subtask(i)'s commit precedes Subtask(i+1)'s

Table 3.2: Edges in the criticality graph.

Table 3.2 describes all the possible edges between the nodes. The first set of edges are those within a subtask (Row 1). They are drawn between the successive stages of a subtask: $S_i \rightarrow E_i$, $E_i \rightarrow F_i$, and $F_i \rightarrow C_i$. The rest of the edges in the table are between subtasks, and represent interactions between subtasks. Specifically, every subtask is started by one of two possibilities. First, it can be spawned by a predecessor, as shown with a Spawn edge $F_i \rightarrow S_j$, where the subtask versions satisfy $i < j$ (Row 2 of Table 3.2). Alternately, the subtask can be squashed by a predecessor and restarted, as shown with a Squash edge $E_i \rightarrow S_j$ (Row 3 of Table 3.2). Once executing, subtasks can also synchronize either from explicit wait instructions or as a result of a dependence predictor. In this case, a *Sync* edge, $E_i \rightarrow E_j$, will be inserted.

A spawned task may be unable to execute in two cases: it does not have a core or context to run on, or it needs to become safe before it can start executing. The latter case may occur, for example, if the subtask has to perform I/O operations that cannot be executed speculatively. The first case is represented with a Resource edge $F_i \rightarrow E_j$ (Row 4 of Table 3.2); the second case is represented with an edge from the Commit node of a subtask to the Execute node of its immediate successor — called BeSafe edge $C_i \rightarrow E_{i+1}$ (Row 5 of Table 3.2).

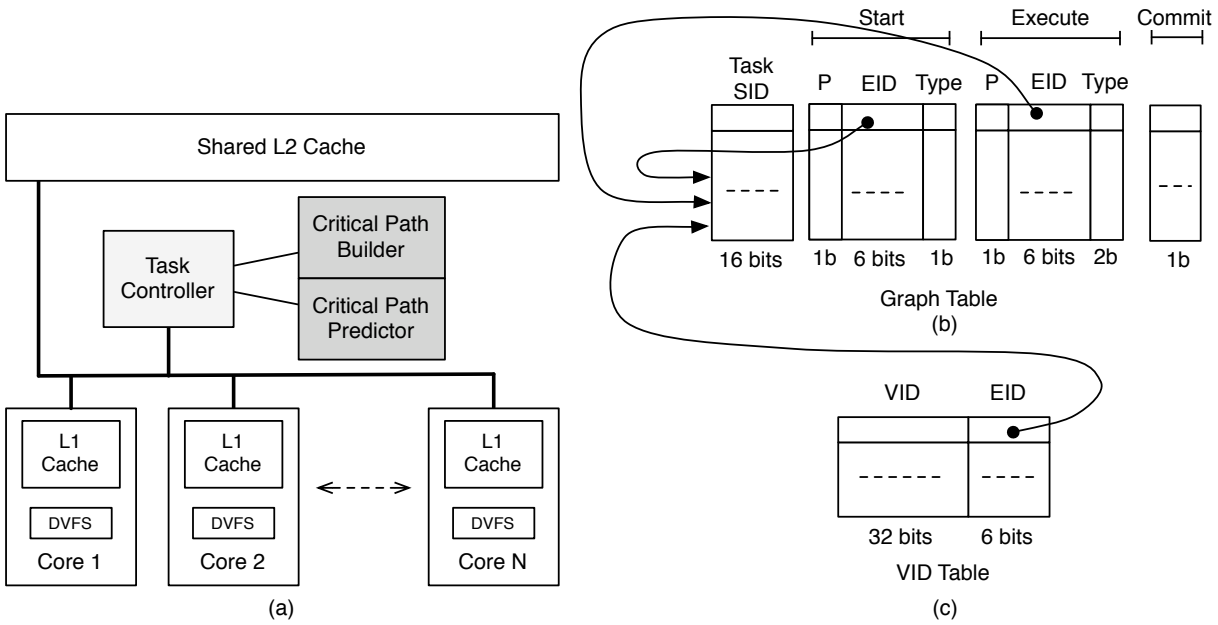


Figure 3.2: An overview of the CAP architecture.Multiprocessor system where CAP is the two dark-shaded modules (a); Graph Table (b); and VID Table (c).

Finally, the fact that each subtask must commit after its immediate predecessor commits is represented with the Commit edge $C_i \rightarrow C_{i+1}$ (Row 6 of Table 3.2). For simplicity, in the remainder of the chapter, the term "tasks" will also refer to "subtasks" in the model.

Figure 3.1(c) shows the graph corresponding to the execution in Figure 3.1(b). The critical path can be computed efficiently using Last Arriving Rules [30] by walking backward from the last *Commit* node in the graph. The critical path in Figure 3.1(c) is highlighted with thicker lines. From this graph, it is clear that *Task B* is not on the critical path and, in fact, could execute longer in a lower power mode without even hurting performance.

3.3 Architecture Design

3.3.1 Overview

This section presents *CAP*, a novel architecture to: (i) build in hardware and analyze the task-level criticality graph dynamically, and (ii) make task criticality predictions based on the graph. These predictions are used to schedule SM tasks power-efficiently in a CMP with DVFS.

Figure 3.2(a) shows a multiprocessor system with a Task Controller (TC) module. The TC keeps track of all running, pending, and finished tasks. It has a queue of *task containers*, each of which has full information on the status of one task. The TC controls task scheduling.

In this environment, CAP adds the two dark-shaded modules: the *Critical Path Builder* (CPB) and the *Critical Path Predictor* (CPP). The CPB builds the criticality graph on-the-fly in hardware, and computes the critical path. The CPP extracts information from the graph needed to make criticality predictions.

When an application initially starts executing, no criticality information is yet available. Eventually, a task commits and the TC forwards a summary of the task's execution, such as who spawned it or who may have squashed it, to the CPB for addition to the criticality graph. When the CPB buffers enough tasks for a meaningful analysis, the critical path is calculated and the most pertinent information from the graph is extracted and saved in the CPP. Then, the CPB is flushed and starts building another graph.

Once the CPP is trained, it predicts one of the executing tasks as critical. To avoid the overhead of too many task preemptions, the CPP only predicts a new task as critical at four events, namely when a task is spawns, squashes, synchronizes, or finishes. When the CPP predicts a new critical task, the TC schedules the critical one to a higher performing core — one with high voltage and frequency. The TC is able to improve the power-efficiency of SM because it is aware of the voltage-frequency values of each core. Consequently, it schedules tasks to cores based on criticality predictions. Since some tasks have slack, they are able to execute more slowly using less power without hurting performance.

3.3.2 Critical Path Builder (CPB)

The CPB is responsible for recording the graph and calculating the critical path. The CPB consists of the two hardware tables shown in Figure 4.5: the *Graph Table* (b) and the *Version ID (VID) Table* (c). The Graph Table provides the storage for the criticality graph. The VID Table provides a mapping from a VID into an index, or Entry ID (EID), in the Graph Table.

Each entry in the Graph Table represents a node of a task in the criticality model. However, only enough information to construct the critical path needs to be recorded, namely which incoming edge is last-arriving for each node. Recall that there are a total of six types of inter-task edges: Spawn, Squash, Sync, Resource, BeSafe and Commit (Section 3.2.2). Since the BeSafe and Commit edges always come from the immediate predecessor, there is no need to record their EID. Hence, only Spawn, Squash, Sync, and Resource edges need to record the source task's EID, and these edges correspond to the *Start* and *Execute* nodes. Since Spawn and Squash arrive at the same node and only one can be critical, only one field is needed to represent the *Start* node's Last Arriving Edge (LAE). Similarly, only one field is needed for the *Execute* node's LAE, but the type field is needed in both cases to identify

which type of edge is last arriving. Each edge is represented by the EID of the task that contains the source node.

So far, this covers the *Start* and *Execute* nodes, but the *Finish* and *Commit* nodes must also be modeled. Since the *Finish* node has no incoming inter-task edges, it need not be modeled explicitly in the table since its LAE is always known by construction. As mentioned above, the predecessor of *Commit* is always known as well, so all that must be recorded is whether the critical path flows from its predecessor's commit or the task's *Finish* node. This requires only a single bit in the table.

The Graph Table also contains information to help the CPP. The P column shown in the figure with the *Spawn* and *Execute* nodes identifies whether the edge was predicted critical during execution. By recording this information in the Graph Table, the CPB can notify the CPP that a non-critical edge was predicted critical. This is valuable for de-training the criticality prediction on an edge in the CPP's hysteresis table.

Now, consider the process of recording the graph. As tasks execute and complete, their information is added to the criticality graph. During execution, the TC assigns each dynamic task a unique name, or Version ID (VID), that can be used for ordering and tracking speculative state. This VID is also used for recording inter-task edges for the criticality graph. During execution of a task, the TC observes all inter-task operations and keeps a record of the Last Arriving edges for node in the task execution model. When a task finally commits, an entry is allocated in the Graph Table for the task, and its VID is set to map to its entry in the Graph Table using the VID Table. Using the VID Table, the EIDs for its incoming edges are identified and added to the appropriate fields.

When the Graph Table is filled with tasks, the critical path is calculated. Since the last-arrival edge for each node is recorded, it is trivial to construct such a critical path. A finite state machine traverses the graph in reverse and constructs the path based on the last-arrival edge information. During the traversal, the critical path information is passed to the CPP to train the predictor.

Thanks to using a task-level criticality model, the CPB is very space-efficient. The storage for the CPB is $34b \times 64$ for the Graph Table plus $38b \times 16$ for the VID Table, for a total of 348B.

3.3.3 Critical Path Predictor (CPP)

The CPP is shown in Figure 3.3 and is organized as a table that records the behavior of critical edges between tasks along with a pointer to the current critical task. The predictor table focuses on the Spawn, Squash, and Sync edges. An entry in the table is accessed using

a hashed value created from the source and destination tasks' Static Task ID (which is a hash of the first instruction address of the task). When an edge is found on the critical path, it is added to the table. If it is already present in the table, a saturating counter is incremented. However, if the edge is calculated as non-critical by the CPB but was predicted critical during execution, the saturating counter is decremented. Each edge predictor is implemented as a 3 bit up/down saturating counter.

In some cases, a predicted critical edge will not appear in the Graph Table because another edge with the same destination node is the LAE. For this reason, every edge predicted critical by the CPP is recorded for each task. When the task commits, it sends that information along with the list of inter-task edges to the CPB. If a predicted critical edge is not the LAE, then the corresponding entry in the prediction table is immediately decremented to reflect that fact.

Each entry in the CPB (Figure 3.4) records the Static Task ID solely for the purpose of training the predictor. To improve the accuracy of the predictor, a more complex hashing function may be used to incorporate additional path history in the Static Task ID.

The CPP generates a new prediction for the critical task whenever the currently predicted critical task spawns, squashes, or syncs. If the predictor table identifies the edge as critical, the other task is predicted as critical. For the case that the current critical task finishes, the current prediction is set to null. From that point forward, all spawns, syncs, and squashes are monitored and checked against the predictor table to find a new critical task.

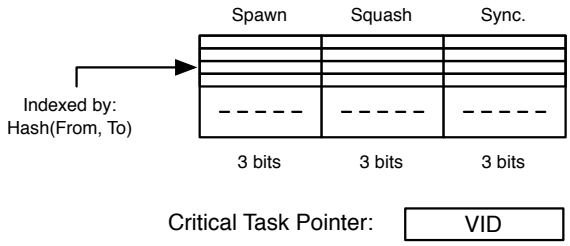


Figure 3.3: Design of Critical Path Predictor (CPP) hysteresis table.

3.3.4 An Example of CAP at Work

Figure 3.4 provides a detailed example of the CAP at work. Part (a) is the same task diagram as the one explained in Figure 3.1. Part (b) shows the graph as drawn by the model of SM execution. Consider each task in (a) as it commits. The first task to commit has a VID of 1 and corresponds to *Sub A.1.* This task is allocated an entry in the Graph Table, namely 29, and the VID table is updated to reflect this mapping. Also, the TC notifies the builder that

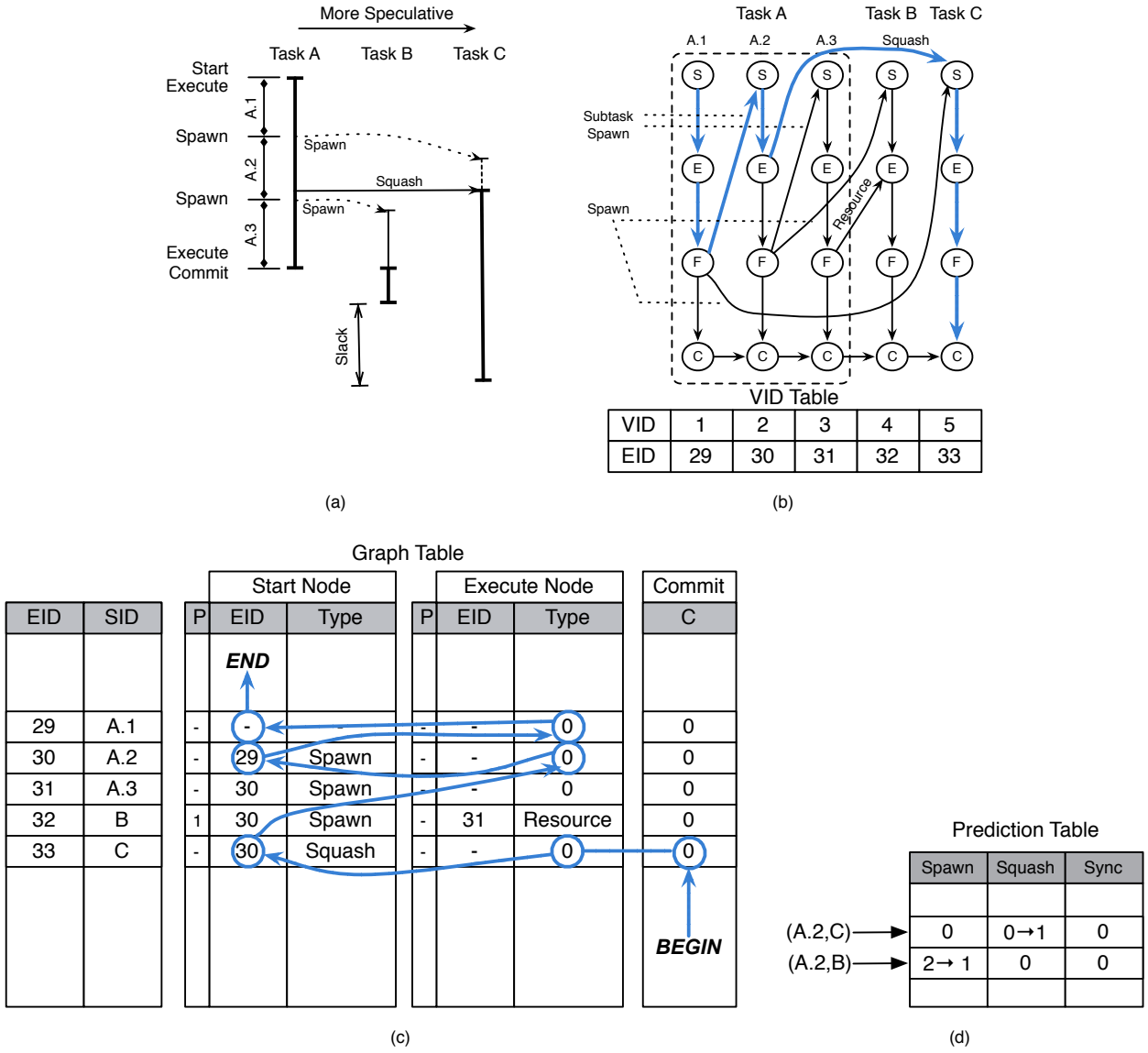


Figure 3.4: An example of the Critical Path Builder and Critical Path Predictor at work during execution.

Sub A.1 spawned *C*. The CPB then fills in each pertinent field in the Graph Table. For this example, presume that Task A is the first one in the system; hence, it has no last arriving edge for its Start or Execute Node since no other task spawned or squashed it. The Commit field is set to 0 to indicate that the Finish node is last arriving.

The next task to commit is *Sub A.2* which is given the next entry in the Graph Table and its fields are updated. It is spawned by *Sub A.1* which is its last arriving edge to its Start Node. Note that the TC provides the spawner in terms of VID. Using the VID Table, it is converted to an EID which points to the appropriate entry in the graph table. The Spawn field acts as a pointer to its parent task in the Graph Table. The Execute field is set to zero

since there are no incoming edges to the Execute Node. Finally, the last arriving edge to the Commit Node is from its own Finish/Start Node. *Sub A.3* proceeds in similar fashion as the previous.

When *Task B* commits, in addition to updating the VID Table and identifying which task spawned it, it must also record a Resource dependence edge, since *Task B* could not execute until *Task A* completed. Also, imagine that when *Task B* was spawned, it was predicted critical; hence, the predicted bit is set in the Graph Table. Finally, *Task C* commits and receives entry 33 in the Graph Table. Since *Task C* was squashed, the Start Node field records the EID of *Sub A.2*. Since *Task C* waits on no other task to commit, its Commit field is also set to 0.

Once the Graph Table is filled, the critical path is calculated by traversing the table in reverse, starting at the last committed task, with EID of 33. For this traversal, the VID Table is no longer needed since all task references are made using EIDs. The last node of the last task is used to seed the calculation, marked by *BEGIN* in the figure. For each node, the CPB tracks backward along the last arriving edges. The dark lines with arrows drawn over the table show the progression of the critical path calculation.

The critical path for these tasks follows the execution of *Task A* until it squashes *Task C*, at which point *Task C* is critical for the rest of the time. Interestingly, *Task B* never appears on the critical path indicating that it has some slack in its execution. While the CPB tracks the critical path, it also updates the Prediction Table. Using the source and destination Static IDs, it locates the appropriate entry to update in the table. For the case of the critical squash, the counter is incremented to reflect that is now believed to be more critical. However, the spawn of *Task B* that was predicted critical is now believed to be less critical, and its prediction is decremented.

Criticality-based Task Scheduling

Since not all tasks are equally important, the criticality-based scheduling algorithm tries to put critical tasks on fast cores and non-critical ones on slow cores. Preemption is employed to move SM tasks between cores in order to benefit from a better voltage/frequency pair. Fortunately, preemption is easy to implement in the SM environment because of the built-in support for state buffering. Each SM task has its speculative state buffered before it commits, hence when a task is preempted, a register checkpoint is sufficient to save the task context.

In addition to scheduling the critical thread on a fast core, the TC also guarantees that the safe thread is always running.

3.4 Methodology

To evaluate CAP on a SM CMP, SESC [65] is used since it provides a cycle-accurate execution-driven simulator with detailed models of out-of-order superscalars, a memory subsystem and a SM protocol. It also includes the models of dynamic power from Wattch [7], Orion [93] and Cacti [73]. Moreover, the simulator is augmented to support per-core DVFS which allows each core to adjust voltage/frequency to a certain operation point independently.

Processor	CAP Parameters		
Frequency: 5.0 GHz @ 70 nm	CPB		
Fetch/issue/comm width: 6/3/3		#Entries	Width
I-window/ROB size: 68/126	Graph Table	64	34b
Int/FP registers: 90/68	VID Table	16	38b
LdSt/Int/FP units: 1/2/1			
Ld/St queue entries: 48/42	CPP		
Branch penalty: 13 cyc (min)		#Entries	Width
BTB: 4K entries, 2-way assoc.	Criticality Predictor	2048	9b
Branch predictor (spec. update):			
bimodal size: 16K entries			
gshare-11 size: 16K entries			

Cache	D-L1	I-L1	L2
Size:	32KB	32KB	2MB
RT:	3 cyc	2 cyc	10 cyc
Assoc:	4-way	2-way	8-way
Line size:	64B	64B	64B
Pend ld/st:	16	-	64
Voltage and frequency pairs			
fast mode	5 GHz	1.6 V	
slow mode	3.5 GHz	1.1 V	
Bus & Memory: DDR-2			
Bus frequency: 533MHz; Bus width: 128bit			
DRAM bandwidth: 8.528GB/s; memory RT: 98ns			

Table 3.3: Processor simulated for CAP experiments. In the table, cycle counts refer to processor cycles.

Two CMP configurations and three scheduling algorithms are considered to measure the impact of criticality analysis. The baseline configuration modeled is a 4 core CMP with SM support, and each core runs at the highest frequency and voltage (identified with *4H*). A power optimized configuration with one core at the highest frequency and voltage along with three at the lowest frequency and voltage (identified with *1H*) is also studied. For each of these configurations, three scheduling algorithms are considered: *Base*, *Sort*, and *CAP*. *Base* implements a naive scheduling in which a task is simply given a free core at random. *Sort* guarantees that the least speculative threads are always running, and, in the case of

1H, with the least speculative (or safe) thread running on the single high-frequency core. *Sort* is based on the common wisdom that "non-speculative tasks are more important than speculative ones." Finally, *CAP* uses the proposed scheme. In all the experiments, all the energy-centric optimizations for SM as suggested in the paper of Renau *et al.* [66] are turned on, including clock gating all unused cores in the SM CMP.

The parameters of the architecture are shown in Table 3.3. Out-of-order cores with an issue width of three are used. Each core has a private L1 cache that stores speculative data using a SM cache coherence protocol similar to [41]. The parameters of CAP are shown in the rightmost column of Table 3.3. The CPB adds about 348 bytes of storage and the CPP adds 2.3KB. Each core runs at one of the two predefined operation points. The fast cores run at 5 GHz and 1.6 V, and the slow cores run at 3.5 GHz and 1.1 V.

The POSH compiler [47] was used to generate the SM binaries for all executions. The binaries are optimized for power [66]. The full SPECint and SPECfp 2000 applications are evaluated with the *Ref* input data set. Exceptions include those applications written in C++ or Fortran, since they are not supported in the infrastructure, and *perlbmk* and *gcc* which fail on the compiler. Also evaluated are a few Olden benchmarks. In the simulations, the initialization (1-6 billion instructions) is skipped, and then execute the applications by about 0.75-1.50 billion sequential instructions.

3.5 Evaluation

In this section, critical paths are characterized, then CAP's impact on performance and power is assessed.

3.5.1 Characterization of Critical Paths

Table 3.4 characterizes the critical path of applications running on *4H-Base*. Note that sub-task edges are not included in this characterization. The first two columns show the application name and the number of dynamic tasks in its execution. The first group of columns labeled *% of Tasks* shows the occurrence of each inter-task edge type as a percentage of the number of tasks. *Squash* and *Sync* edges are far more common in the SPECint applications, while *Resource* edges are more common in the SPECfp and Olden applications due to more abundant parallelism. The presence of *Resource* edges indicates that useful work is often waiting to execute.

The second group of columns shows the percentage of the critical path contributed by each inter-task edge type. *Spawns* are significant fraction of the critical path for all three kinds

Application	Tasks	%Sq	%Dep	%Res.	%SSafe	%Sp	%Sq	%Dep	%Res.	%Com	%SSafe
bh	20015	0.3	0.0	92.5	1.4	18.1	0.2	0.0	81.2	0.1	0.4
em3d	79891	0.7	100.0	4.1	0.1	11.4	0.5	85.8	2.2	0.0	0.0
health	395310	0.6	0.0	1.3	32.7	67.0	0.2	0.0	0.7	5.4	26.7
mst	1386969	0.2	0.0	1.8	0.2	98.8	0.2	0.0	1.0	0.0	0.0
perimeter	51598	1.1	0.6	19.6	0.0	63.5	2.6	0.6	33.3	0.0	0.0
art	1410753	0.5	0.0	25.0	1.2	88.3	0.1	0.0	11.3	0.0	0.3
equake	375276	2.4	1.5	16.3	0.4	91.9	1.1	0.2	6.7	0.0	0.0
mesa	239590	0.2	100.0	3.4	0.7	23.4	0.0	76.3	0.1	0.0	0.0
bzip2	886632	9.5	6.2	2.1	0.4	88.0	7.3	2.2	2.1	0.0	0.4
crafty	663757	25.1	79.8	0.0	0.0	44.5	14.7	39.3	1.4	0.0	0.0
gap	1335206	41.2	100.0	2.7	18.3	25.1	25.0	29.0	2.6	0.1	18.2
gzip	2495855	0.9	42.3	0.0	0.0	78.3	0.8	20.8	0.0	0.0	0.0
mcf	18402007	1.7	50.6	1.5	0.0	64.8	1.3	32.7	1.2	0.0	0.0
parser	5006683	16.7	63.5	2.7	0.2	59.2	13.3	25.4	2.0	0.0	0.2
twolf	6183991	19.6	38.7	6.2	0.0	39.3	23.0	32.0	5.8	0.0	0.0
vortex	1847494	1.3	18.3	6.4	0.6	77.5	1.5	9.8	10.2	0.1	0.8
vpr	174134	51.9	29.8	29.9	0.8	27.5	42.0	11.1	18.3	0.1	1.0
Olden-G.M.	135237	0.5	0.0	7.0	0.0	38.7	0.4	0.0	5.3	0.0	0.0
CFP-G.M.	502446	0.6	0.0	11.1	0.7	57.5	0.0	0.0	2.0	0.0	0.0
CINT-G.M.	1922929	8.8	37.4	0.0	0.0	51.3	7.2	17.5	0.0	0.0	0.0

Table 3.4: Characterization of the critical path.

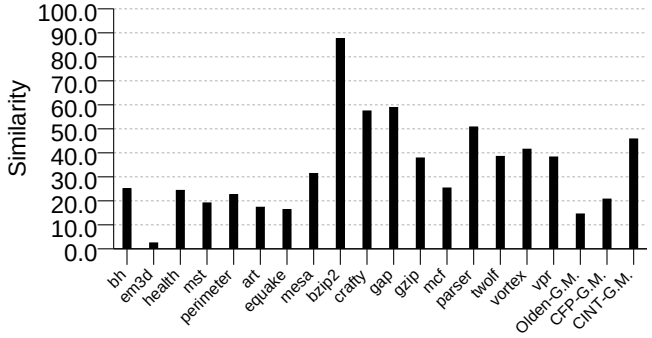


Figure 3.5: Similarity between the instructions on the critical path and the ones on the safe path.

of applications, though more significant for SPECint. Not surprisingly, the SPECints have many critical *Squash* and *Sync* edges, whereas the Olden and SPECfp have many *Resource* edges on the critical path. Note that *Commit* and *BeSafe* edges are rarely on the critical path. Since the overwhelming majority of critical edges are from *Spawns*, *Squashes*, and *Syncs*, this supports the decision to focus on these three edges in the CPP (Section 3.3.3).

The similarity between all the instructions on the critical path and the ones on the safe path running on *4H-Base* is shown in Figure 3.5. The similarity is defined as follows:

$$Similarity = \frac{|I_{crit} \cap I_{safe}|}{|I_{crit} \cup I_{safe}|}$$

where I_{crit} denotes all the instructions on the critical path, and I_{safe} denotes all the instruc-

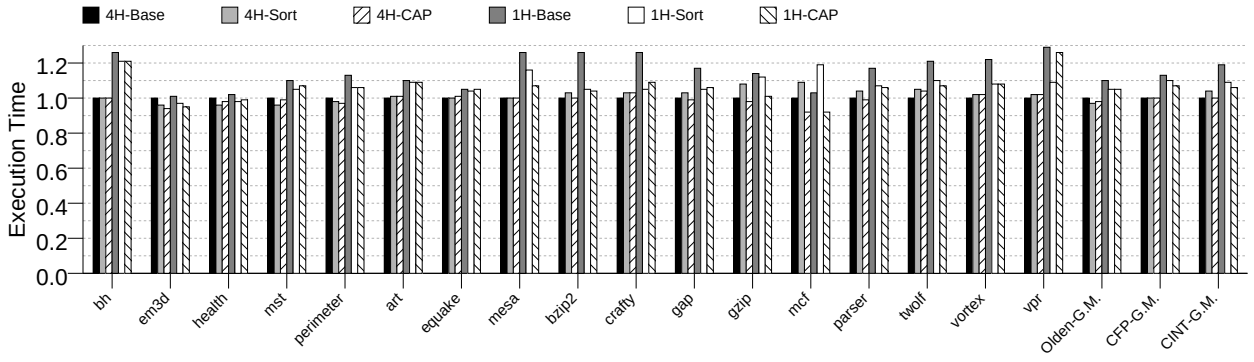


Figure 3.6: Normalized execution time to *4H-Base*.

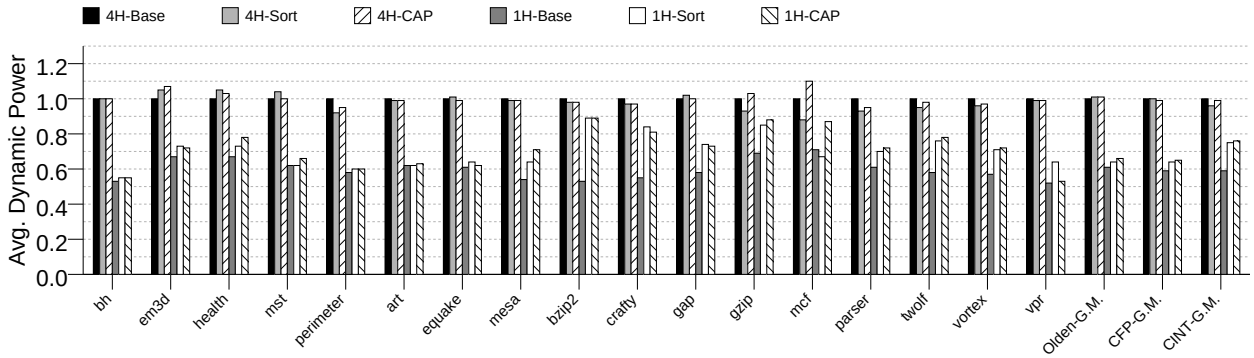


Figure 3.7: Normalized average power to *4H-Base*.

tions on the safe path. On average, the similarity between the critical path and the safe path is roughly 14% for Olden, 20% for SPECfp, and 46% for SPECint. The Olden applications show the least similarity between the safe and critical path. Since these applications have few squashes, the critical path is usually speculative. However, the relatively higher squash rates of SPECints result in a higher similarity between the safe path and the critical path.

3.5.2 Performance and Power Impact

Now consider the performance and power impact of criticality analysis for SM. Figure 3.6 shows execution time of six configurations normalized to *4H-Base*. Overall, the execution times of the *4H* configurations are similar, although a few applications, like *em3d* and *mcf*, do see some benefit from CAP scheduling on this configuration. The *1H* configurations perform worse than the *4H*. However, of these configurations, *1H-CAP* performs the best with a performance degradation of only 6%, 7%, and 5% for SPECint, SPECfp, and Olden respectively.

Average dynamic power is significantly reduced as shown in Figure 3.7. For the most part, average power for each set of configurations is similar. *1H-CAP* reduces the average

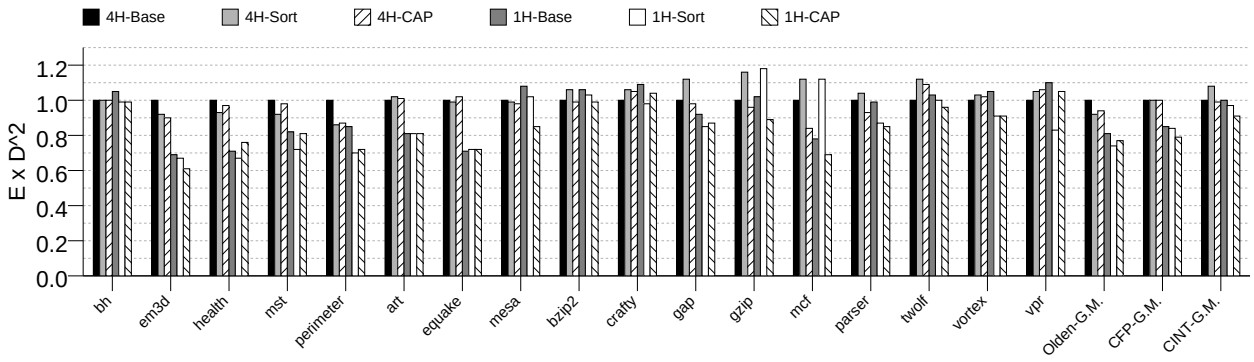


Figure 3.8: Normalized energy delay-squared to *4H-Base*.

dynamic power compared to *4H-Base* by 24%, 35%, and 34% respectively. *1H-Base* posts the lowest average power, however, it is also the worst performing configuration.

Figure 3.8 shows $E \times D^2$. For SPECint and SPECfp, *1H-CAP* offers the lowest $E \times D^2$ with a reduction of 9% and 21% relative to *4H-Base*. *1H-Sort* also provides a good reduction in $E \times D^2$ and is much better than *1H-CAP* for *vpr*. However, *1H-Sort* has very poor behavior for *gzip*, *mcf* (61% better with *1H-CAP*), and *mesa* since these applications have few squashes leading to many more speculative tasks being on the critical path.

3.5.3 Accuracy of Critical Path Prediction

The accuracy of the critical path that CAP calculates is dependent on two key factors: (i) how accurately the CPB approximates the critical path of the program, and (ii) how accurately the CPP chooses critical edges. Figures 3.9 and Figure 3.10 quantify these two effects, respectively.

Figure 3.9 shows how accurately the builder labels a task as critical during execution. This figure is calculated by recording a trace of the execution of the program that is annotated with the calculations made by the CPB at runtime. Then, the full trace's critical path is calculated. The fraction of the critical path identified by the CPB as compared to the actual critical path is plotted for each application. Overall, the accuracy is quite high for all the applications, achieving well over 90% accuracy for SPECint and SPECfp. The Olden average is low because of *bh*, which experiences some pathological behavior with the small Graph Table (only 64 entries) used to calculate the critical path.

Figure 3.10 shows the accuracy of the predictions made by the CPP. Again using a trace of the execution, this plot shows the fraction of edges on the critical path predicted correctly. The prediction accuracy varies considerably from one application to another. For the applications that benefit most from *CAP* like gzip and mcf, the prediction accuracies are

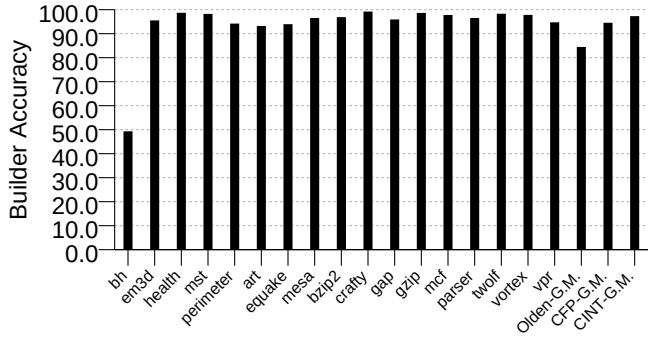


Figure 3.9: Accuracy of the CPB in calculating the critical path.

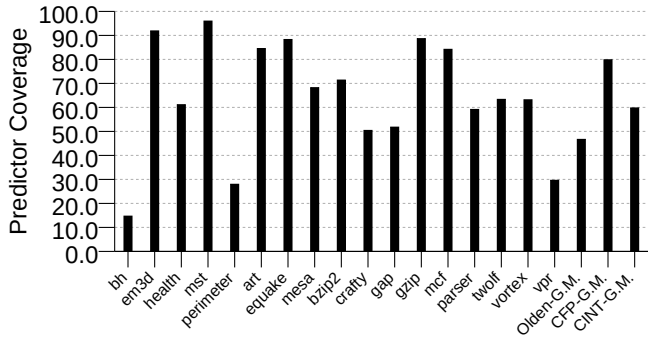


Figure 3.10: Accuracy of the CPP in predicting critical edges.

high. However, high prediction accuracy does not guarantee that *CAP* will provide energy savings. If there is little slack in the application, then even high accuracies will not prevent the application's performance from being hurt. On the other hand, low accuracies do not mean that *CAP* does not help. Mesa is able to achieve significant reduction in $E \times D^2$ despite a lower overall prediction accuracy. However, mesa does have a few sync edges with high prediction accuracy which provide its gains.

3.5.4 Discussion: What limits CAP's effectiveness?

These results show that *CAP* is able to reduce $E \times D^2$ beyond what is capable by *Sort*. However, there are still many applications which do not significantly benefit from *CAP*. There are a couple of key factors that may be limiting CAP's effectiveness.

The first factor is resource limitations. Several of the applications have many resource

edges on the critical path for significant fractions of their execution. There is very little that CAP can do in these situation since CAP exploits slack in the application. CAP needs some degree of load imbalance to provide benefit.

The second factor is the difficulty of predicting dependences at runtime. These experiments already leverage a state-of-the-art dependence predictor, yet many squashes still happen. Since these squashes are difficult to predict, the CAP predictor will anticipate them poorly resulting in poor scheduling decisions. This effect is quite severe in vpr which has a task that suffers from many late squashes — the task is 90% through its execution before it is squashed. Improving dependence behavior either through prediction or better task selection will significantly improve CAP's effectiveness.

3.6 Summary

Speculative Multithreading (SM) on a Chip Multiprocessor (CMP) has been proposed as an effective technique to speed-up hard-to-parallelize applications. However, aggressive speculation can be power inefficient. To improve SM's power-efficiency, this work made three contributions.

First, it developed a widely-applicable, novel task-criticality model for SM. A hardware implementation of the model is stored in a special on-chip module and is refined as execution proceeds. Second, the work proposed the CAP architecture, which (i) uses the model to analyze and predict the criticality of tasks in a SM application at run-time, and (ii) uses criticality to schedule tasks on a SM CMP with per-core DVFS for power-efficient execution. Critical tasks are scheduled on fast cores, while non-critical ones run on slower ones.

Experiments with SPECint, SPECfp and Olden applications show that CAP reduces the average dynamic power of an optimized baseline by 24%, 35%, and 34% respectively, while degrading performance by only 6%, 7%, and 5% on average. Furthermore, it reduces $E \times D^2$ by 9%, 21%, and 23% respectively. Compared to scheduling based on task ordering, CAP can reduce $E \times D^2$ by as much as 61%. Finally, the task criticality composition of different applications was characterized.

Overall, while power efficiency is a challenging problem in SM, CAP has shown that dynamic criticality-based task scheduling can reduce power consumption substantially with small performance overhead.

Chapter 4

SoftBulk: Software Exposed Bulk Operations

4.1 Introduction

Recent proposals for Transactional Memory ([8, 14, 97]) and Thread-Level Speculation ([14]) have called for *Signatures* in hardware to accelerate memory disambiguation. The power of signatures lie in their ability to represent a set of addresses concisely while allowing for set operations directly on the signatures. Instead of operating on one address at a time, sets of addresses are disambiguated in aggregate. Signatures have been so effective at reducing the overheads of conflict detection, employing them in other contexts may provide commensurate benefits. Potential uses for signatures are easily inspired by looking at known uses for memory disambiguation hardware.

One potential use for signatures is hardware support for Runtime Disambiguation ([3, 31, 44, 58, 61, 80]). Nicolau first pointed out that further optimization of code was possible if some disambiguation checks were moved to runtime. Since then, software ([3, 80]) only and hardware supported mechanisms ([31, 61]) have been studied for various code optimizations. Of these techniques, the Memory Conflict Buffer ([31]) is particularly reminiscent of signatures because it provides hardware for disambiguation that continually checks for conflicts between an earlier scheduled load and a later store, alleviating software from the burden of inserting comparisons. Instructions inserted by the compiler notify the MCB of speculative loads and check whether any conflicts ever occurred.

Inspired by techniques like the MCB that allow for compiler directed disambiguation, SoftBulkis proposed as an architecture for exposing signatures and bulk operations directly to software through the instruction set architecture (ISA). SoftBulk makes two key additions to the architecture: (i) instructions for accumulating addresses into signatures and for operating on them efficiently, and (ii) Bulk Register Files that are part of the architectural state of the processor. Using SoftBulk, programs can collect information about their own memory access patterns and use that information in many ways.

To show the potential for SoftBulk, it is applied to function memoizatoin. Memoization is a way of caching the results of a computation so that it need not be repeated–in this sense, it

is a kind of dynamic redundancy elimination. For arbitrary functions which may have implicit memory inputs or produce side effects, memoization is usually not possible. However, using SoftBulk, a memoization algorithm is designed for an arbitrary function in C/C++. The algorithm will be referred to as Signature Enhanced Memoization, or MemoiSE for short. MemoiSE leverages SoftBulk to record which memory locations are read and written into a signature and performs disambiguation on that signature. As a result, MemoiSE can easily test whether implicit inputs or side effects have been changed since the memoized call.

SoftBulk can also be used for many other optimizations and in support of other systems. Transactional Memory (TM) and similar speculative execution environments can potentially benefit from SoftBulk since it provides efficient support for disambiguation. In fact, many of the operations described in [8, 97] can be provided using SoftBulk. Furthermore, debuggers often disambiguate sets of addresses; for example, watchpoints are a common mechanism that detect a write to a specified address. A few watchpoints are typically supported in hardware, but once they are exhausted, adding more comes at a considerable performance cost. SoftBulk, on the other hand, could watch a large number of addresses with minimal cost. The potential applications for SoftBulk are numerous.

This work makes three contributions. First, it proposes exposing signatures and signature operations in the ISA and details the important aspects of this interface. Second, it describes in detail the architecture for SoftBulk. Thirdly, it provides a detailed example use of SoftBulkcalled MemoiSE. MemoiSE is unique in that it does not require static dependence analysis to correctly memoize arbitrary functions.

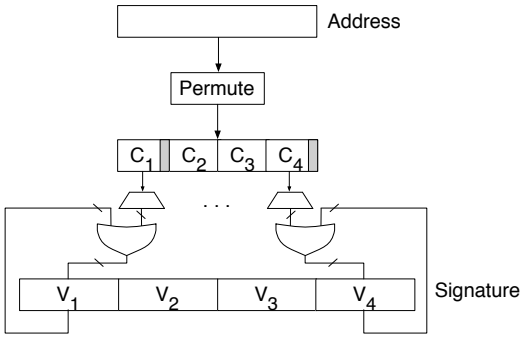
This paper is organized as follows: Section 4.2 is a brief background on bulk disambiguation; Section 4.3 presents the SoftBulk software interface; Section 4.4 details the SoftBulk implementation; Section 4.5 describes Signature Enhanced Memoization; Section 4.6 evaluates MemoiSE; and Section 4.7 presents related work.

4.2 Bulk Signatures and Operations

Bulk ([14]) is a set of hardware mechanisms that simplify the support of common operations in an environment with multiple speculative tasks such as Transactional Memory (TM) and Thread-Level Speculation (TLS). A hardware module called the Bulk Disambiguation Module (BDM) dynamically summarizes into Read (R) and Write (W) signatures the addresses that a task reads and writes, respectively. Signatures are $\approx$2 Kbit long and are generated by accumulating addresses using a Bloom filter-based [5] hashing function (Figure 4.1(a)). Therefore, they are a superset encoding of the addresses. When they are communicated,

they are compressed to ≈350 bits.

The BDM also includes units that perform the basic signature operations of Figure 4.1(b) in hardware. For example, intersection and union of two signatures perform bit-wise AND and OR operations. The combination of the decoding (δ) and membership ($\in$) operations provides the *signature expansion* operation. This operation finds the set of lines in a cache that belong to a signature without traversing the cache. It is used to perform *bulk invalidation* of the relevant lines from a cache.



(a)

Op.	Description
$\cap$	Signature intersection
$\cup$	Signature union
$= \emptyset$	Is signature empty?
$\in$	Membership of an address in signature
δ	Signature decoding into cache sets

(b)

Figure 4.1: Signature encoding (a) and primitive operations (b).

Bulk has been proposed for environments with speculative tasks such as TM or TLS that perform conflict detection between tasks only when a task tries to commit [14]. Speculatively-written lines are kept in-place in the cache and cannot be written back before commit. Speculatively-read lines can be displaced at any time because the R signature keeps a record of the lines read.

In this environment, signature operations are simple building blocks for several higher-level operations. As an example, consider the commit of task C. The BDM in the task's processor sends its W_C to other processors. In each processor, the BDM performs *bulk disambiguation* to determine if its local task L collides with C, as follows: $(W_C \cap R_L) \cup (W_C \cap W_L)$. If this expression does not resolve to empty, L should be squashed. In this case, the BDM performs bulk invalidation of the local cache using W_L, to invalidate all lines speculatively written by L. Finally, irrespective of whether L is squashed, the BDM performs bulk invalidation of the local cache using W_C, to invalidate all lines made stale by C's commit. In the whole process, the BDM communicates with the cache controller, and the cache is completely unaware of whether it contains speculative data; its data and tag arrays are unmodified.

Several other systems have adopted signatures and bulk operations for the purpose of disambiguation. Proposals for Transactional Memory [8, 97] have added signatures in hardware

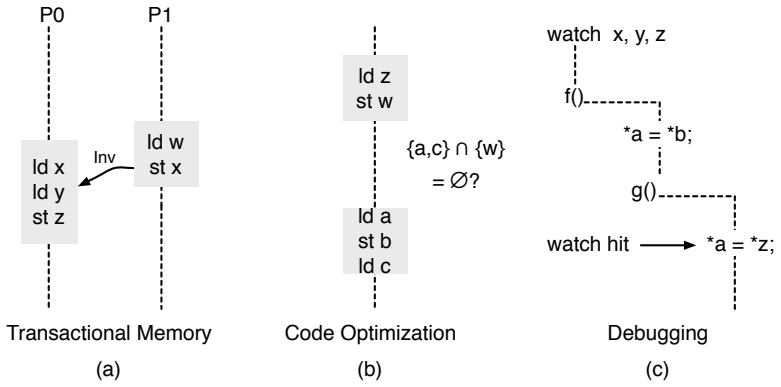


Figure 4.2: Three potential uses for SoftBulk.

to speedup disambiguation between threads. BulkSC ([12]) proposed using bulk signatures to enforce sequential consistency.

4.3 SoftBulk: Software Exposed Bulk Operations

The goal of SoftBulk is to expose signatures and bulk operations to software to support the design of optimizations and systems that can benefit from coarse-grained disambiguation support in hardware. To that end, the ISA of a superscalar processor is extended with support for bulk operations and provide a register file for holding signatures. This section focuses on the software interface and the key architectural supports needed to support it.

4.3.1 Motivating Examples

To motivate the design of the software interface, consider the following three systems: TM, a compiler, and a debugger. Figure 4.2 shows three scenarios that can leverage bulk disambiguation. Part (a) shows an TM that is executing two transactions in different threads. Each transaction's accesses are encoded in a signature and used for disambiguation.

Part (b) shows the results of a compiler analysis in which it has identified two regions of code that it is unable to fully analyze. Using signatures, the compiler can generate code that tests such conditions for entire regions of code at a time. Note that it is similar in concept to the MCB, but works on code regions instead of individual instructions.

Part (c) shows a debugger watching a few addresses. Although some processors do have limited support for watchpoints, it is typically very few and watching more comes at a high overhead. Signatures, on the other hand, can encode a large number of addresses with high accuracy. A debugger could use a signature in hardware to effectively provide near

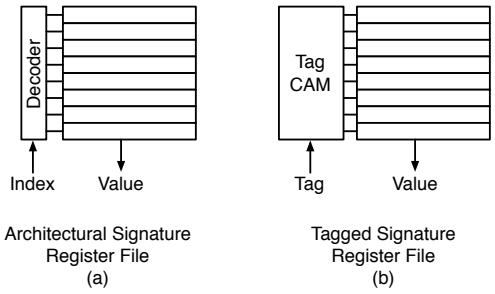


Figure 4.3: A typical register file (a), versus a tagged file (b).

unbounded watch support.

Each of these systems are quite different in their use and end goal, and yet their operations, at least in part, can be represented in a relatively small set of bulk operations. Keeping these examples in mind, the remainder of this section details the characteristics of a software interface for SoftBulk.

4.3.2 The Software Interface

Basic Software Support

In its most general form, a signature is a special, low-level data type for representing supersets of addresses. Whether exposed to the programmer or not, such a data type needs a few basic supports to be useful in a variety of settings. Table 4.1 lists these instructions first. They include a load and store for moving signatures in and out of memory, respectively, and a move instruction. In addition, the empty instruction provides a test for the empty set, and S0 is a signature register that always holds an empty signature.

Collecting a signature

Encoding the elements in a signature is a common operation. In parts (a) and (b) of Figure 4.2, identifying the set of addresses operated on during a window of execution is very common. In TM for example, each load/store is logged in a signature. The interface may support logging these addresses in a signature in one of two ways: (1) an instruction which inserts a single address into a signature, or (2) a set of instructions that begin and end *collection* of addresses into a signature. For flexibility, (1) is desirable. For efficiently handling common behavior, like the critical section in Figure 4.8, (2) is also important since it eliminates significant overhead. Hence both are supported in this interface.

Collecting a signature over a region of code suggests an interface somewhat different from typical architectural registers. General purpose (GP) registers, for example, have their values

Category	Operations	Description
Basic Software Support	ld S1,Addr	Initialize from memory
	st S1,Addr	Store to memory
	mv S1,S2	Move from one reg to another
	empty R1,S1	$R1 \leftarrow (S1 == S0)$
	S0	Empty signature
Collection	bcollect.(rd,wr,rw) *tag*	Collect all following operations into TSR[*tag*]. Depending on the specifier, only reads,writes, or both.
	ecollect *tag*	Stop collecting on TSR[*tag*].
	union S3,S1,S2	$S3 \leftarrow S1 \cup S3$
	insert_elem R1,S1	Hash encode R1 into S1.
	filtersig *tag*,R1,R2	Do not collect addresses between R1 and R2 for TSR[*tag*]
Disamb- iguation	bdisamb.(loc,rem) *tag*	Continually disambiguate TSR[*tag*] with local/remote accesses. For local, all following operations are disambiguated. For remote, disambiguation begins by the time a following memory op is executed.
	edisamb.(loc,rem) *tag*	Stop disambiguation on TSR[*tag*]. All previous operations will be disambiguated and completed.
	intersect S3,S1,S2	$S3 \leftarrow S1 \cap S3$
	member R1,R2,S1	$R1 \leftarrow (R2 \in S1)?1 : 0$
	sigstatv R1,*tag*	Return Status Vector for TSR[*tag*] in R1. Guarantees all pending operations for TSR[*tag*] have been completed.
Persistence	allocsig R1,*tag*	Obtain register in TSR File and name it *tag*, return Status Vector in R1.
	dallocsig *tag*	Deallocate TSR[*tag*].
	movsig.(rd,wr) *tag*,S1	Copy contents of TSR[*tag*] into S1, and Guarantee that all pending operations on TSR[*tag*] have completed.
Checkpoint and Exception	checkpoint *tag*,*target*	Create a checkpoint. If checkpoint aborts, jump to *target*.
	rollback	Rollback checkpoint.
	commit	Commit checkpoint.
	exptsig *tag*,*target*	Except to *target* if a conflict occurs on TSR[*tag*].

Table 4.1: Bulk Software Interface.

saved and restored from the stack when a subroutine is called. However, if a subroutine is called during collection, the addresses accessed by the subroutine should also be added. One way to support collection across arbitrary control flow is with registers that remain *persistent* in a special register file. Instead of providing architectural names, these registers can be allocated on demand by software and given a unique name, or tag, for reference. Figure 4.3 illustrates a typical register file and a tagged register file, in this case the Tagged Signature Register (TSR) File. Instead of being indexed by a particular index, a TSR has a unique name specified in software.

Table 4.1 show the two instructions, *bcollect* and *ecollect*. Both take as their single argument a *tag* that identifies a register in the Tagged Signature Register File (TSRF). For now, a tagged register is assumed to be always available. In addition, the *insert_elem* instruction may be used for explicit collection. *union* provides a way of combining signatures together, as described in Section 4.2.

Disambiguation

Like collecting a signature, disambiguation is a very common common operation. Disambiguation identifies whether two sets of addresses share a common element. It is challenging because a conflict may occur frequently, and sometimes it is unknown where in the code a conflict may actually occur. This advocates a few different modes for disambiguation support.

Figures 4.4 (a)-(d) depict these cases. Figure 4.4(a) shows the case that a compiler has identified two regions, the shaded part of the program, for disambiguation. Note that while the regions are apart, the compiler is able to analyze all the code in between and prove that it will not cause a conflict. As a result, the compiler can schedule a specific disambiguation test at the appropriate location. This simply requires the `intersect` instruction.

Part (b) is a similar example, but the compiler does not know when a conflict may occur because it could not adequately analyze the code. In this case, the compiler can collect the accesses during the intervening region and explicitly test for any conflicts. This solution is general but may not work well in some cases since false positive rates dramatically increase with set size. Since it may not know the length of execution, collecting the region in another signature could lead to significant aliasing. For long regions of code, a better approach allows the hardware to monitor for a conflict, potentially one instruction at a time This case is shown in part (c). Disambiguation support is provided in hardware and enabled using *bdisamb* and *edisamb*. Both take a *tag* as an argument identifying a register in the TSR File. These instructions direct hardware to continually check active operations for conflicts

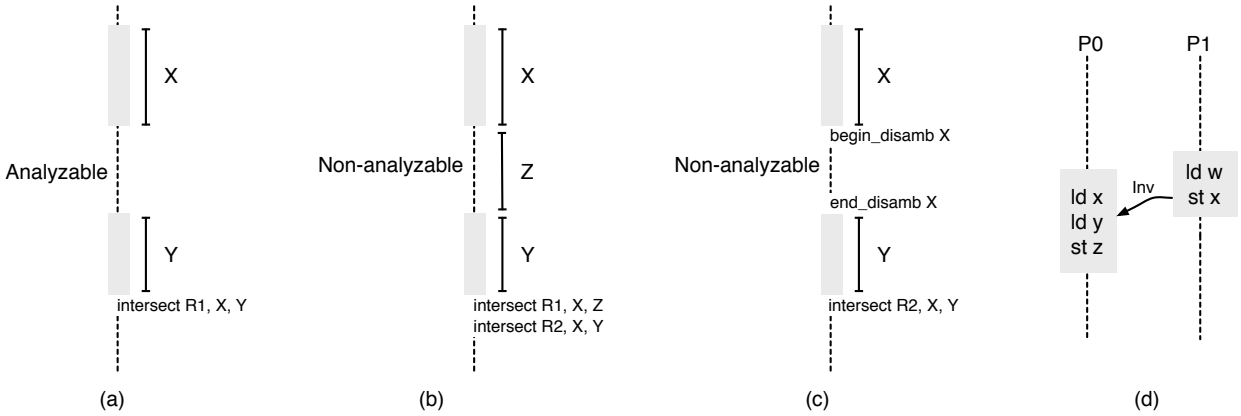


Figure 4.4: Four examples of disambiguation using SoftBulk.

with a signature stored in the named register.

Figure 4.4(d) (a duplicate from Figure 4.2) shows the classic case of TM. Two critical regions execute concurrently on two different processors. Hardware support to detect a conflict from a remote thread enables standard TM operations. Therefore, this interface supports automatically detecting a conflict on a signature even by coherence operations triggered by other threads. The `bdisamb` and `edisamb` instructions support disambiguation with respect to the local or remote memory access streams.

Other instructions that support disambiguation are the `sigstatv` instruction, which detects if a conflict or deallocation has occurred on a tagged register, and the `member` instruction which tests a single element for membership in a signature.

Managing Tagged Registers

The `allocsig` and `dallocsig` allocate and deallocate a tagged register, respectively. Each one takes a *tag* as an argument that binds a name to a dynamic instance of a signature register. Because it is allocated with a name, it may persist across arbitrary control flow.

Tagged registers are not guaranteed to remain allocated. A context switch or running out of free registers may result in deallocation. Also, because tagged registers may be deallocated without warning, the `sigstatv` instruction is instrumental in using them properly. Figure 4.7 shows the status vector that is returned by the `sigstatv` instructions.

Because the signature registers are divided into two different files, the `movsig` instruction ferries signatures from tagged registers into the Architectural Signature Register (ASR) File.

Checkpoints and Exceptions

Checkpoints are an essential part of many kinds of speculative execution like TM and TLS. SoftBulk can integrate seamlessly with checkpointing, and provide efficient, new functionality since hardware similar to SoftBulk can be used to provide versioning in the cache.

In support of checkpointing, `checkpoint`, `rollback`, and `commit` are added to control checkpoint creation, rollback, and commit. Checkpoints can be referenced in the same way as tagged registers, allowing the contents of their associated signatures to be manipulated and used for computation with other signatures.

Also, to provide for eager conflict detection, the `exptsig` instruction registers an exception handler for a given tagged register. When a conflict is detected on that register, a precise exeception handler is triggered. If no exception is ever registered on a tagged register, then no action happens on a conflict, even for the case of checkpoints. This allows full programmer control over checkpoint creation, rollback and commit.

4.3.3 Semantics of SoftBulk Instructions

The SoftBulk ISA offers a rich set of features for disambiguation. Providing a clear semantics for each of these instructions is critical for them to be employed correctly. The semantics of these instructions derive from two sources: (i) the semantics of bulk operations and the superset encoding of signatures, and (ii) the allowed order of memory operations for a given program order.

Operations on Signatures Signatures are supersets, and as such, have certain properties. For example, if an element has membership in a signature, it may be because it was actually inserted or an artifact of the superset encoding (a false positive). On the other hand, if it does not have membership, then it is certain that the element is not in the set. Because disambiguation shows the absence of one set in another, it is calculated with certainty using supersets. Therefore, it is paramount to know signatures are not exact sets and have different properties of use when constructing operations on them.

Collecting and Disambiguating Addresses in Signatures The collect and disambiguate operations work on the sequential memory access stream. Collection guarantees that at least those addresses that are accessed after *begin* and before *end* will be included in the signature. Disambiguate provides a similar guarantee for memory accesses local to a processor. Conflicts will be detected as if all operations were executed sequentially.

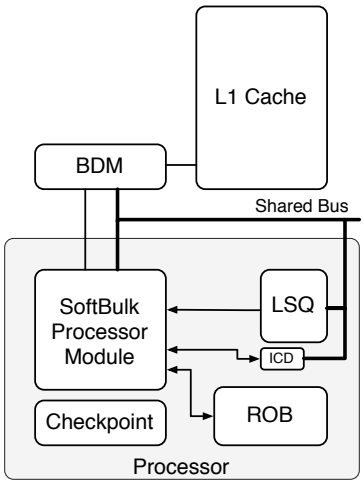


Figure 4.5: SoftBulk Architecture.

The primary goal of remote disambiguation is to detect if a conflict has occurred with respect to another core. Remote disambiguation must work as expected in two cases: (1) the signature has already been collected, and (2) the signature is currently being collected. The first case is trivial: remote disambiguation will begin and end when the *bdisamb* and *edisamb* are executed. The second case must work as the programmer expects: any conflict for the signature being collected will be detected. This requirement has an important implication for the architecture and will be discussed in Section 4.4.

4.4 SoftBulk Implementation

To support SoftBulk, several extensions are made to a superscalar processor. Figure 4.5 provides an overview of the proposed extensions. The central part of this proposal is the *SoftBulk Processor Module* (SPM) which provides most of the support for collecting, disambiguating, and otherwise operating on signatures. The SPM interacts with the Re-Order Buffer (ROB) and Load/Store Queue (LSQ) in support of collection and disambiguation. The In-flight Conflict Detector (ICD) is a new structure that aids disambiguation. Checkpoint, rollback, and commit are provided by the SPM in conjunction with the Bulk Disambiguation Module (BDM) [14] and the Register Checkpoint.

The SPM implements most of the functionality of SoftBulk. Figure 4.6 provides a detailed view of the module. Each part of the SPM and its interactions with the rest of the processor are covered in more detail in the following sections.

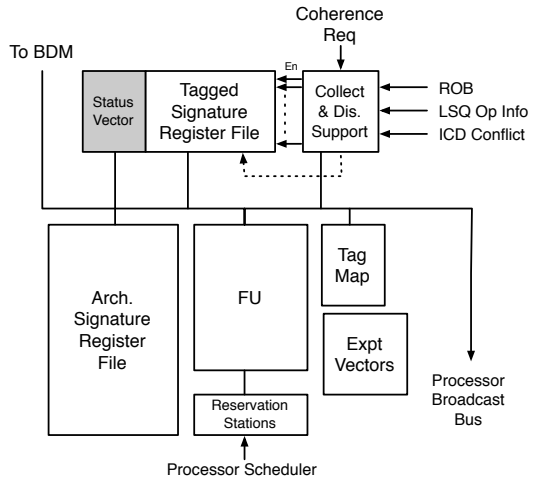


Figure 4.6: SoftBulk Processor Module.

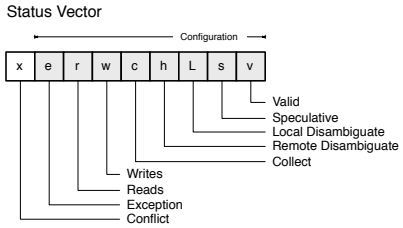


Figure 4.7: Status Vector associated with each Tagged Signature Register.

4.4.1 Signature Register Files and Functional Unit

The SPM provides two signature register files. The Architectural Signature Register (ASR) File are directly accessible by the compiler through an architectural name. The Tagged Signature Register (TSR) File uses a *tag* to identify a register. Both files are shown in Figure 4.6. The ASR file is implemented as a typical register file using a direct indexed SRAM array. The TSR file is implemented in two parts: (1) a fully associative tag array mapping a tag to an index called the Tag Map, and (2) an SRAM array storing the signature.

Associated with each TSR is a Status Vector which records configuration information and current status. Figure 4.7 shows the vector and its fields. Each field is 1 bit. The lower eight fields return configuration information and the upper bit indicates if there has been a conflict. The `sigstatv` instruction returns this entire vector.

The Functional Unit (FU) of the SPM, shown in Figure 4.6 supports operations on the ASR File. Bulk Operations are issued to the SPM's Reservation Stations. Operations on the ASR File are performed by the Functional Unit and can occur as soon as their data dependences are available. These operations include the following instructions: `mv`, `empty`,

`union`, `insert_elem`, `intersect`, and `member`. The ASR File may be renamed and operate just like the other register files with regard to renaming and scheduling logic. Operations involving a TSR are handled differently, as discussed in the following sections.

4.4.2 Collection, Disambiguation, and Operations on Tagged Registers

Operations on TSRs form the cornerstone for SoftBulk. Supporting them efficiently with only minor architectural changes is of great importance. The most important TSR operations are collection and disambiguation; hence, the architecture for operating on TSRs is designed to optimize for those operations.

Collection, disambiguation, and other operations involving TSRs are scheduled in a stage immediately prior to or during retirement. Scheduling at retirement avoids two complications of O-o-O execution: (1) rolling back state on a branch misprediction, and (2) knowing exactly which loads occur between the *begin* and *end* instructions despite re-ordered execution. By performing collection and disambiguation late in the pipeline, few changes need to be made to other structures.

Collection and Local Disambiguation

Given late execution of these operations, collection and local disambiguation are relatively simple to envision. In the case of collection, once the `bcollect` instruction is executed, all following accesses are added to the appropriate TSR. Since they are near retirement, the address of each memory access can be streamed in order from the LSQ. Collection ends when the `ecollect` instruction is executed. Local disambiguation happens in a similar way, but instead of inserting the addresses into a signature, they are used for conflict detection.

All local memory accesses are streamed into the SPM while it is actively collecting or disambiguating. The streaming of addresses is achieved via interaction between the SPM, LSQ, and ROB. When a `bcollect` or `bdisamb` is issued, it waits at the SPM until it reaches the retirement stage, only then is it performed. At this point it executes and notifies the LSQ to begin sending the address, type of access, and ROB index of all memory operations as they retire. As they are streamed into the SPM, they are handled by the module labeled *Collect & Dis Support*. The logic found in this module is shown in Figure 4.8. Using the configuration bits from the Status Vector of each TSR, it is determined whether the current input address should be added to the signature, or if it should be disambiguated with respect to the signature. Also, note that part of the enable process is ensuring that the address is in

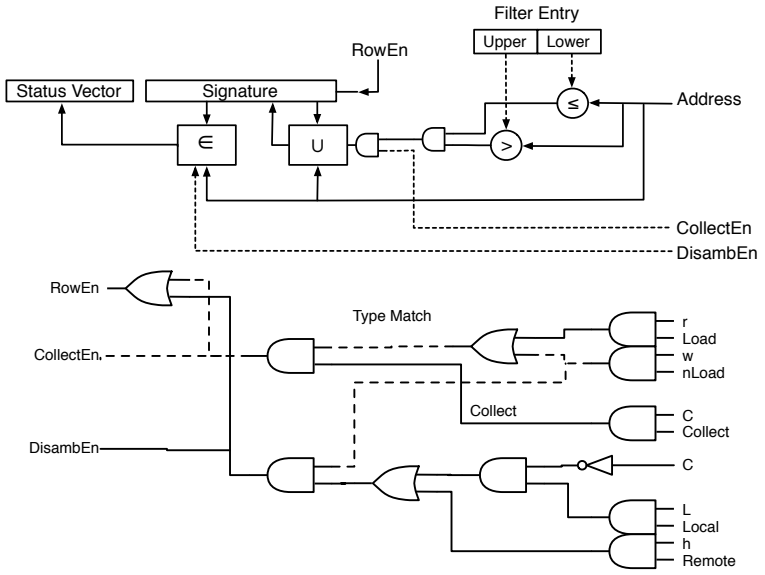


Figure 4.8: Logic for collection and disambiguation of addresses.

the appropriate range for collection, as set by the `filtersig` instruction. If no conflicts are detected on an address, the ROB is notified that the instruction can retire; otherwise, an exception may be generated (discussed more in Section 4.4.4). The logic shown in Figure 4.8 will be needed for each tagged register.

Remote Disambiguation

Remote disambiguation detects a conflict between a TSR and remote coherence requests. These coherence requests are sent directly to the SPM for disambiguation, just like local requests. However, remote disambiguation is more problematic than local. To illustrate why, consider the execution trace in Figure 4.9. Here, while a signature is being collected, it is also being disambiguated with respect to remote accesses. In this case, after the load is executed an invalidate is observed. However, because the load is collected and disambiguated much later (just before retirement), the earlier invalidate does not generate a conflict when it is serviced by the SPM. While this may be allowed by the consistency model, it would be incorrect given the semantics of the disambiguate instructions.

To catch this case, logic is added to watch for conflicts on in-flight instructions. In this way, a conflict on an instruction that has not yet retired is never missed. The logic added to detect this case is called the In-flight Conflict Detector (ICD). The ICD contains a counter-based Bloom filter that keeps a record of all in-flight operations. When an operation executes, it is added to the Bloom filter, and when it retires, it is removed. When `bdisamb.rem` is decoded, the ICD is enabled and begins recording all memory accesses in its Bloom filter.

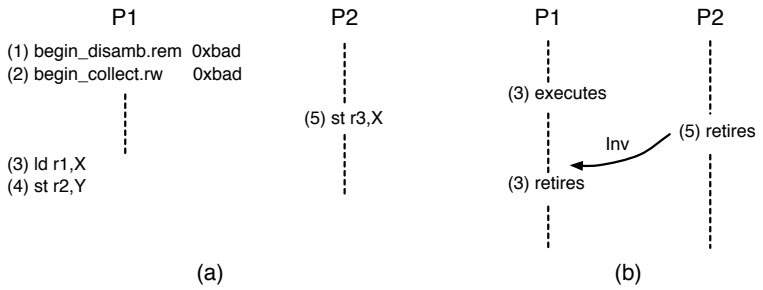


Figure 4.9: An example of simultaneous disambiguation and collection which erroneously misses an invalidate.

If a remote coherence request is found to be encoded in the Bloom filter, the ICD sets a flag indicating a conflict and remembers the ROB index of the youngest instruction in the LSQ. All TSRs that are collecting and performing remote disambiguation while this youngest instruction is still in-flight will have their Conflict bit set in their Status Vector. Once the youngest instruction retires, the ICD clears its conflict flag. The ICD must remain active from the time the first `bdisamb.rem` is decoded until the last matching `edisamb.rem` is retired and no signatures are in the remote disambiguate configuration.

4.4.3 Checkpointing

The extensions for checkpointing are provided as described by Ceze et al ([14]). The register checkpoint takes a snapshot of the architectural registers when the checkpoint instruction retires, and the BDM supplies versioning in the cache. Checkpoints are integrated with the SPM by providing the signatures in the BDM with tags, just like the TSR File. The only difference is that checkpoints do not need to pre-allocate the tag (using `allocsig`) since the checkpoint instruction performs that action. If the checkpoint cannot be allocated, control branches to *target* as specified in Table 4.1.

While the contents of the signatures in the BDM are not allowed to be manipulated directly using the collect or disambiguate instructions, the signatures can be moved into the ASR for operations. Once in the ASR, the signatures can be placed in another TSR and used for further collection and disambiguation.

If a TSR was allocated before a checkpoint, it is allowed to persist regardless of the success or failure of the checkpoint. However, if a TSR is allocated during a checkpoint that is later aborted, it is automatically discarded during rollback. The *Speculative* bit is set in the Status Vector for any TSR allocated in a checkpoint. This bit identifies which TSRs to deallocate.

4.4.4 Exceptions

Exceptions may be triggered when a conflict is detected on a TSR. The SPM supports registering a table of exception handlers. If one is set for a given TSR, it is called on a conflict. A bit in the Status Vector indicates whether such an exception is registered. For a local conflict, a precise exception is generated for that instruction. For the case of a remote conflict, the handler is called as soon as detected by the ICD.

If an exception is requested, the SPM notifies the processor's front end of the target address and informs the ROB of the excepting instruction so that it can flush it and all earlier instructions. The exception handler takes the typical actions to guarantee proper recovery: it pushes the return address into a register and disables handling of additional exceptions. Since other TSRs may still be under disambiguation, additional exceptions are buffered and serviced sequentially.

4.5 MemoiSE: Signature Enhanced Memoization

Memoization ([52]) is a common approach for replacing a redundant, or precomputed, call with its outputs. For many programming languages, especially C/C++, automatic memoization is often impossible because dependence analysis cannot guarantee that inputs from memory will not change. Using SoftBulk, a general memoization transformation is implemented with low overhead, allowing any function to memoize correctly. In the remainder of this section, the correctness of the memoization algorithm is discussed as well as how to perform the transformation and some optimizations to reduce its overheads.

A General Memoization Algorithm

Before describing the memoization algorithm, some basic terms must be established. A function call is specified by its *call signature*: a unique name, explicit input arguments, and explicit output arguments. However, these are not the only inputs and outputs. A function may have implicit inputs in the form of non-local variables that it references, and it may have implicit results it returned as changes to non-local variables (side-effects). In addition, functions sometimes have implicit local variables carried from one invocation to the next.

Memoization algorithms work by caching the inputs and outputs of a function in a lookup table. When the function is invoked, the lookup table (or solution table) is searched for an entry with an identical set of inputs. If such an entry is found, the outputs (or solution) are copied out of the lookup table into the appropriate locations (memory or registers) instead

of executing the function. Explicit inputs and outputs are easily managed in such a lookup table.

To build a generic memoization algorithm, implicit inputs and outputs must be taken into account. However, a conservative assumption can make general memoization tractable using SoftBulk. Instead of ensuring the equivalence of all implicit input values, it is enough to guarantee that implicit inputs have not changed since a previous execution. A signature encodes all the implicit input addresses and hardware disambiguates against this signature during execution. If no conflicts are discovered, it is safe to assume the inputs have not changed. Implicit outputs (side effects) can be handled in a similar way. A signature can encode the output addresses. Using disambiguation, changes to these outputs can be detected. As long as they remain unmodified, no log of implicit writes is necessary for memoization.

During execution of the function, it is possible that an implicit output overwrites an implicit input. Since an input is changed, the function invocation cannot be memoized. This case will be referred to as *internal corruption* and must be detected to guarantee that memoized results are truly deterministic.

Now, memoization can be extended to general functions by associating a tagged signature with the inputs and outputs of a function. The memoization test must prove the following for an entry in the lookup table: (1) the explicit inputs must match an entry, (2) the implicit inputs from memory are unchanged as determined by a TSR, (3) implicit outputs to memory are also unchanged as determined by a TSR, and (4) the implicit memory outputs do not overwrite any of the inputs.

4.5.1 MemoiSE Implementation

Figure 4.10 shows the code needed to test whether a call is redundant with respect to a previous memoized call. The main elements of the algorithm are shown in the figure: (a) the layout of the code, (b) a solution table statically allocated per function, (c) a prologue to test for replacement, (d) a memo setup phase that starts collection and disambiguation of the input and output sets, and (e) an epilogue that finalizes an entry in a solution table for use on the next invocation.

Prologue The `foo_prologue` shown in Figure 4.10(c) ascertains whether the call can be memoized. The prologue traverses the solution table looking for a matching entry. The first criteria checked is validating the implicit inputs and outputs, which requires three checks: detecting if the allocated TSR still exists, has encountered no conflicts, and is not currently

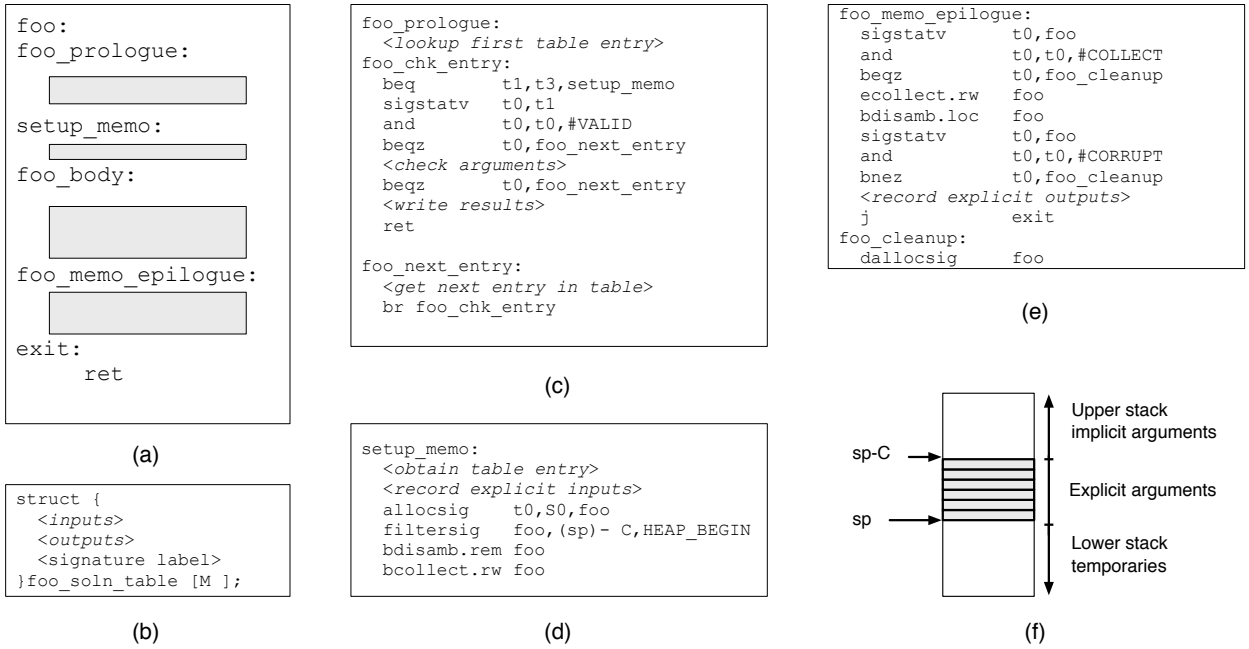


Figure 4.10: Code for the MemoiSE algorithm on function *foo*. Shown above are (a) overall layout of the function, (b) the statically allocated solution table, (c) prologue that searches for a valid solution, (d) setup in the event a solution is not found, (e) epilogue that finalizes the solution table entry or does clean up, and (f) a illustration of which addresses are filtered during collection.

in use (due to recursion). All of these are accomplished by checking the Status Vector for the appropriate tagged register. Each solution needs a unique name, so the address of the entry is used as the tag. Lastly, the explicit input arguments are compared against the values in the table. If a match is found, the explicit outputs are provided from the table and the function returns. Otherwise, another entry in the table is searched.

Memo Setup If the function call is not memoized, the collection phase initializes the necessary structures to test for redundancy on the next invocation. This involves two key operations: preserving register inputs and allocating a tagged register to collect the memory inputs and outputs of the function call.

During the execution of the function body, the TSR collects all the implicit inputs and outputs. However, some accesses to memory are filtered from the TSR, namely those accesses that are temporaries for a single invocation. These stack locations are shown in Figure 4.10(e). Note that some of the stack may be used for explicit inputs. Because these are checked by value, accesses to these memory locations are not implicit and should be excluded. Also, any access to the stack in the function's call frame, the *lower stack*, is a temporary created for the current invocation and will not survive past the return. There-

fore, they are also excluded. The range of addresses beginning with any explicit arguments extending to the lower stack are identified using the `filtersig` instruction.

Epilogue The `foo_memo_epilogue` code makes the final steps of memoizing the execution in the solution table. This code takes a few simple steps. First, it terminates collection in the TSR. Next, it checks to make sure the TSR is still allocated and did not encounter internal corruption. If both of these checks pass, the explicit outputs of the function are added to the solution table and local disambiguation begins on the TSR. Otherwise, the TSR is deallocated.

4.5.2 Selection and Optimization

Since only some functions can benefit from memoization, a profiler should identify which functions are amenable to memoization and only apply the transformation in those cases. Furthermore, the overheads of memoization can be too large to make it profitable even when a function is frequently redundant. Optimizations that eliminate or reduce some sources of overhead can broaden the applicability of the technique.

Function Selection

Functions selected for memoization should effectively trade-off memoization overhead for eliminated computation resulting in a net performance gain. This trade-off can be studied by analytically describing the relationship between average run length, the number of explicit inputs and outputs, the fraction of memoized invocations, and the overhead for memoizing a call to the function.

Consider a function, f that is called N times and was memoized N_M calls, with an average run length of I_f when it is memoized. Equation 4.1 calculates the number of instructions saved due to memoization, and Equation 4.2 calculates the overhead of memoization. $O_{prologue}$, O_{setup}, and $O_{epilogue}$ calculate the instruction overhead from the prologue, setup, and epilogue code shown in Figure 4.10.

$$I_{Saved} = N_M \times I_f \tag{4.1}$$

$$I_{Ovhd} = N_M \times O_{prologue} + N \times (O_{setup} + O_{epilogue}) \tag{4.2}$$

Instructions are saved only when calls are memoized, but each invocation of f incurs overhead. While the exact cost of a given invocation depends on the path taken, each of the

overheads can be estimated as follows:

$$O_{prologue} = m_{avg} \times (P_{in} \times c_{in} + P_{out} \times c_{out} + c_t) \quad (4.3)$$

$$O_{setup} = c_{alloc} + P_{in} + (P_{in} \times c_{in} + c_t) \times m_{max} \quad (4.4)$$

$$O_{epilogue} = c_{epi} + P_{out} \times (1 - r_{ic}) + c_{clean} \times r_{ic} \quad (4.5)$$

where m_{avg} is the average number of lookups in the solution table, m_{max} is the maximum size of the solution table, P_{in} is the number of explicit inputs to the function, P_{out} is the number of outputs, c_{in} is the overhead of comparing each input, c_{out} is the overhead of copying each output to or from the table, c_{alloc} is the constant overhead of allocating a TSR, c_{epi} is the constant overhead of working with the TSR in the epilogue code, c_{clean} is the constant overhead of deallocating a TSR, and r_{ic} is the ratio of internal corruption.

When I_{saved} is set equal to I_{Ovhd}, an expression for the minimum function size can be expressed as, $I_{fmin} = O_{prologue} + \frac{1}{r_M} \times (O_{setup} + O_{epilogue})$, where r_M is the fraction of memoized calls to f. The minimum size of a function is largely determined by how frequently it can be memoized and the cost of searching the solution table for a match. Looking more closely at I_{fmin}, each overhead is significantly dependent on either the number of inputs, outputs, or both.

Static analysis of a function can identify P_{in}, P_{out}, and, in many cases, if internal corruption is guaranteed, but reasonable values for average execution length and ratio of memoization are difficult to predict statically. Fortunately, SoftBulk can serve as an effective profiler to identify the opportunity for memoization. Also, function run length is easily estimated using a traditional profiler. Altogether, a profiler could effectively select only those functions that are most likely profitable.

Optimizing Search of the Solution Tables

Since the overhead of memoization could be prohibitive for small routines, it is important that the solution table not be the source of that overhead unnecessarily. From Equations 4.3-4.4, it is clear that m_{max} and m_{avg} are a function of the solution table size and the cost of searching the table, respectively. A smaller table reduces O_{setup} and $O_{prologue}$ since fewer entries are searched, and a smarter search would reduce $O_{prologue}$. From the evaluation it will be clear that many functions need only a single entry in their solution table to extract most of the available redundancy. Therefore, a single entry solution table is employed for functions exhibiting such behavior. The result is that some smaller functions are able to benefit from memoization, and there are less overheads per invocation.

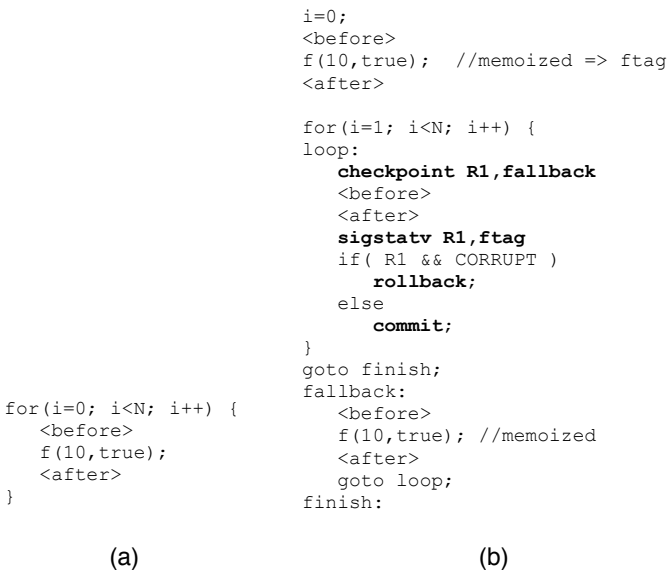


Figure 4.11: Example of Checkpoint-based Call Elimination in action: (a) shows the original code, and (b) the result of optimization.

Call Site Pruning

Functions are often called from many different locations within a program. Each call-site's input parameters may differ in predictable ways, making some call sites more likely to memoize than others. For those call sites that never contribute to reuse and are themselves never redundant, MemoiSE can be disabled. The result is less wasted overhead, which makes the beneficial cases more profitable.

Checkpoint-based Call Elimination

In some instances of memoization, the probability that a call is memoized is very high. In these cases, it is desirable to remove even the overhead of performing the lookup and simply assume it succeeds. This could remove $O_{prologue}$ for a large fraction of invocations in some cases studied. With the help of checkpointing, such an optimization is possible.

Figure 4.11 shows an example code snippet to illustrate this case. Part (a) shows the original loop calling a function with two constant parameters and some implicit inputs and outputs. In (b), the one iteration is peeled off to memoize the call. In the main loop, the call to f is omitted from the loop, but each loop is protected by a checkpoint. Before the iteration can complete, it checks to make sure that `f`'s tagged signature has no conflicts and if so commits. If a conflict has occurred, the iteration is rolled back, and executes the fall back code at label `fallback`. The overhead in this loop is less than that needed to lookup

a solution, and it can be amortized by unrolling several iterations of the loop into a single checkpoint. However, this should be done with care so that the rollback penalty is not too great and to ensure that the state generated by the loop does not exceed the hardware's storage capacity.

4.6 Evaluation

4.6.1 Setup

Pin is used to evaluate the potential of SoftBulk and MemoiSE. Pin is a software framework for dynamic binary instrumentation which offers the flexibility to study a wide variety of applications. To evaluate the usefulness of MemoiSE, instruction count rather than execution time is used to evaluate this system.

Using the Pin analysis tool, detailed information for production quality applications are collected and analyzed for memoization opportunities. Table 4.2 shows the applications analyzed. Three of these are well distributed applications found and used on many personal computers. Supertux is an open-source arcade game, and SESC is an architectural simulator. For each application, over 1 billion instructions were collected and analyzed using Pin.

4.6.2 Memoization Opportunity

Figure 4.12(a) shows the normalized dynamic instruction count for each of the applications using the Signature Enhanced Memoization (MemoiSE) and all optimizations against a run without MemoiSE. The average (geometric mean) reduction in instructions that are memoized is 8.8%. This average is the result of two sources: the instructions saved due to memoization (11.2%) and the overhead of the transformation (2.4%). As shown for *gaim*, the fraction of overhead can be significant (5.8%).

Figure 4.12(b) characterizes the extent of the opportunities for memoization given the application binaries studied. *Useful* are those instructions that were selected for memoization because they afforded a positive reduction in instruction count even when considering their associated overhead. *Loss* identifies the fraction more that were too costly. There is room to improve the memoization technique by 2%, on average. Furthermore, note that for *gaim* and *supertux* the opportunity for memoization is close to 30% of the dynamic execution. However, this is not the limit of opportunity. Many functions that are candidates for memoization were inlined by the compiler. This effect is evaluated later for *sesc*.

Application	Experimental Setup
firefox	**Description:** A popular web browser. **Analysis:** Begins after initialization while it loads the I-ACOMA webpage.
gaim	**Description:** An open source instant messaging program. **Analysis:** Begins once a client is running and consists of opening a new message window,sending a message, and receiving a message.
ooimpress	**Description:** OpenOffice presentation software. **Analysis:** Begins with opening sample.ppt and continues while a user interacts with it.
sesc	**Description:** SESC is an architectural simulator available from SourceForge.com. **Analysis:** Performed on a simulation of the application *mcf* using the default configuration available with distributed with SESC.
supertux	**Description:** A 2D "jump'n run" arcade sidescroller game like Mario Brothers. **Analysis:** Performed during game play, beginning when the penguin drops to the ground, and continuing until a large enough sample is collected.

Table 4.2: Applications studied and their experimental setup.

4.6.3 Optimizations

In Section 4.5.2, several optimizations were described that have the potential to either increase opportunities for redundancy, or reduce costs of overhead. So far, the results have assumed the presence of these optimizations. Figure 4.13 shows the results of each optimization.

In the figure, optimizations are labeled as follows: all optimizations (A), profiling in conjunction with the optimized Solution Table (ST), profiling in conjunction with call site pruning (CS), profiling alone (P), and a naive compilation which selects every function that is memoized at least once (N). The quantities are normalized to the instruction count of the baseline without MemoiSE. Each bar is divided into two quantities. The portion above zero are the *Gains* from memoization as a percentage of the dynamic instruction count without

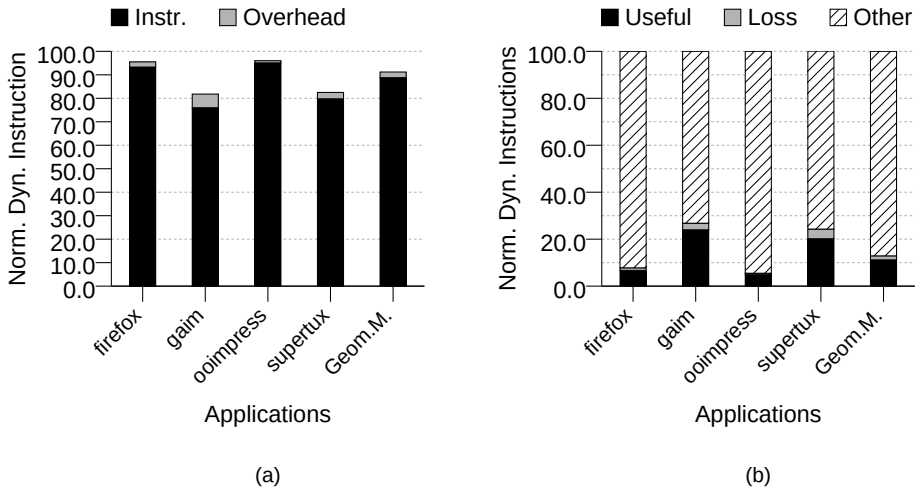


Figure 4.12: Normalized dynamic instruction count (a), and all possible memoized instructions (b).

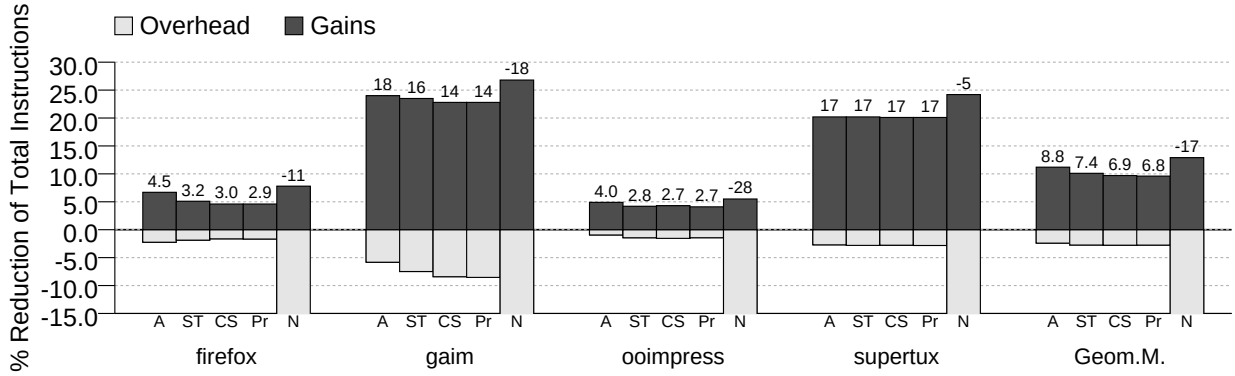


Figure 4.13: Impact of optimizations and profiling.

any optimization. The *Overhead* is below zero and is the cost of memoization and lookup. The number shown above each bar is the difference between the Gains less the Overhead, showing the total savings in instructions for using the optimization. A better optimization will have a taller *Gains* bar, a shorter *Overhead* bar, or both.

Overall, the average reduction is highest when all optimizations are employed. As each optimization is removed, some reduction is lost either due to fewer gains or greater overheads. Applied independently, ST works better than CS at removing overheads and discovering additional redundancy. However, together they do better than either one can alone. Of all the optimizations, profiling is by far the most important since it prevents intolerable overheads for functions that rarely benefit from MemoiSE. Without any optimization, the

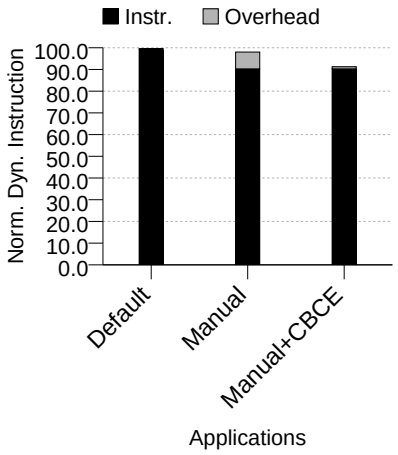


Figure 4.14: The benefits of MemoiSE on sesc using manual changes to the program. *Default* is an optimized compile of sesc (using -O3). *Manual* changes the code to prevent inlining of a few functions. *Manual+CBCE* modified an important loop using Checkpoint-Based Call Elimination.

average normalized instruction count increases by 17%.

A Case Study on SESC

An analysis of *sesc* compiled with -O3 using gcc resulted in very few opportunities for MemoiSE. However, if sesc is compiled with no optimization, far more opportunities are discovered. The discrepancy was not due to elimination of the functions statically but because they were inlined and not detectable by the analysis tool. Therefore, a few profitable functions were prevented from being inlined to provide a comparison.

The first two bars in Figure 4.14 evaluate the outcome of MemoiSE on an optimized build of sesc (Default) compared with one manually altered (Manual) to prevent inlining in a few cases. The result is that MemoiSE offers a slight reduction in instruction count.

Careful analysis of the overhead for Manual showed that a single, small function provided most of the overhead and gains. The call is made from a loop that dominates execution time. Furthermore, its parameters never change, however it has implicit inputs that change rarely. Given these characteristics, it is ideal for Checkpoint-Based Call Elimination, as described in Section 4.5.2. The last bar in Figure 4.14 shows the result of CBCE manually applied to this loop. Most of the overhead is eliminated, providing a significant reduction in instruction count of 8.8%.

4.6.4 Characterization

To further understand how the instruction count reduction is achieved, one important function from each application is listed in Table 4.3(a). Part (b) shows the dynamic characteristics for each function in (a), respectively. The second column shows the number of calls in the sample analyzed by Pin; the next column shows the average dynamic size in instructions of each invocation. The fourth column shows the percentage of calls that were memoized. The fifth column shows the execution weight of the total execution attributed to all invocations of that function. The next column shows the estimated savings per invocation (includes all calls to the function), and the final two columns show the average number of accesses that were reads or writes, respectively, in TSRs allocated for a function.

App	Return Type	Name	Explicit Ins/Outs
firefox	`PRInterval-Time`	`PR_MillisecondsToInterval`	`PRUint32 milli`
gaim	`void`	`pango_font_get_glyph_extents`	`PangoFont *font, PangoGlyph glyph, PangoRectangle *ink_rect, PangoRectangle *logical_rect`
ooimpress	`sal_Bool`	`SfxObjectShell::IsReadOnly`	`this`
sesc	`bool`	`OSSim::enoughMTMarks1`	`this,int pid,bool justMe`
supertux	`void`	`Sector::collision_static`	`this,collision::Constraints* constraints, const Vector& movement, const Rect& dest, GameObject& object`

(a)

App.	#Calls	Size	%Red	%FP	%Weight	SavPerCall	Rset	Wset
firefox	80834	93	99.6	0.	0.6	72.7	4.0	0.0
gaim	840131	281	42.7	0.	27.7	62.0	20.0	0.0
sesc.sp	33704106	35	100.0	0.	21.4	28.9	2.0	0.0
ooimpress	2352	414	95.7	3.0	0.2	393.	50.0	0.0
supertux	29464	5027	29.0	0.	12.0	4784	552.	0.0

(b)

Table 4.3: Five important functions from the applications considered with their call signature (a) and runtime characterization (b).

Firefox `PR_MillisecondsToInterval` is a frequently called function that accounted for 0.6% of the dynamic execution monitored. Nearly all of its instances are memoized. This function converts milliseconds into clock ticks, and is most often called with a constant parameter. Dynamically, it has many of the characteristics of a pure function, except that it reads on average four values from memory to perform part of the conversion. These values could change, but, more commonly, are initialized at the beginning of execution based on the platform and never change again.

Gaim This function gets the logical and ink extents of a glyph within a font. The properties of a glyph within a font do not change and are requested frequently in Gaim as each character is processed. Memoization is enhanced as characters are repeated. Therefore, larger solution tables are desirable for this function. Overall the function is not large with only 281 instructions and has four explicit inputs for comparison, but it still saves 62 instructions on average per memoized invocation.

OOImpress This function checks the status of an object. Because it is labeled const, it never updates the objects it references. `SfxShellObject` appears to be a common base class used for many objects in the program. Since it has only one explicit argument for comparison and has substantial reuse, the savings per call are large.

SESC SESC monitors several conditions to determine when it should begin and end detailed simulation. While the conditions are optimized, they are performed frequently–some kind of check is required after each instruction is executed. While the function is only 35 instructions, this adds up over the entire execution, especially in the beginning phase of execution which executes instructions without performing simulation.

Supertux Part of the game logic in Supertux is detecting when collisions occur. `Section::collision_static` is part of this logic, and is called from three sites in the same function, `Section::collision_static_constraints`. The first two locations are in loops, with each iteration changing one of the input parameters. As a result, neither of those call sites are reused. However, between the second and third sites, it is frequent that no change occurs to the parameters, allowing reuse.

4.7 Related Work

4.7.1 Signatures

SoftBulk builds on the burgeoning body of work using hardware signatures for efficient disambiguation. Bulk ([14]) was the first paper to propose the idea of signatures and was followed by several other proposals. LogTM Signature Edition (SE) ([97]) proposes signatures for recording speculative state in lieu of extending cache line state. It also leverages the signature for disambiguation between transactions that are fully contained in hardware. SigTM ([8]) introduces a hybrid TM system that accelerates conflict detection and isolation

in hardware using signatures but relies on software for transctional versioning, commit, and rollback.

BulkSC ([12]) proposed using signatures to enforce sequential consistency. Threads in a program are divided arbitrarily into speculative chunks and their read and write sets are stored in signatures and used for disambiguation. A centralized arbiter performs disambiguation among all threads and ensures the conditions for sequential consistency are never violated. Chunks are allowed to commit after they are disambiguated with respect to all other threads without a violation of consistency.

4.7.2 Disambiguation and Speculative Optimization

SoftBulk can provide Runtime Disambiguation (RTD). Nicolau [58] first proposed Run-time Disambiguation (RTD) as a way to complement static dependence analysis. Optimized code was generated on the condition that a disambiguation test evaluated as unaliased, however, Nicolau's scheme did not rely on any hardware support. Many other software schemes for runtime disambiguation have been proposed [3, 80].

The Memory Conflict Buffer (MCB), by Gallegher et al [31], identified Nicolau's disambiguation tests as a source of overhead and replaced the explicit tests with hardware that performed the checks automatically, allowing efficient speculative optimization. The MCB recorded a hash of a speculative load's address and checked it against a hash of all committing stores. If a conflict is found between a load and a store, fixup, code is called and executed to repair the program state. Interestingly, the MCB uses conservative conflict detection assumptions similar to SoftBulk: if an entry is displaced from the MCB or in the case of a false hash conflict, a conflict is conservatively assumed.

SoftBulk can also be used to record the addresses of speculatively executed loads and perform disambiguation automatically with respect to intervening stores. Furthermore, signatures in SoftBulk can hold many load or store addresses simultaneously, allowing collection and disambiguation over larger regions of code.

Neelakantam et al ([57]) propose using hardware atomicity for reliable software speculation. In their scheme, atomicity is provided by hardware similar to Transactional Memory that takes a checkpoint and buffers program state in case a rollback is required. Within the atomic region, various optimizations can be applied, however, assertions must be left behind in the code to guarantee that the transformation is correct. MemoiSE leverages similar support for Checkpoint-based Call Elimination which allows a call site to be removed provided none of the intervening code changes the result the function would compute. These conditions are easily specified using SoftBulk, however similar support is not available in [57].

4.7.3 Memoization of Functions

Michie [52] first proposed memoization as a general way to avoid computing redundant work, and it is routinely applied in dynamic programming [21] and functional programming languages.

Memoization has been studied at the granularity of instructions [45, 46, 74, 75] and coarse-grained regions [20, 36, 69, 96]. Sodani et al [75] empirically characterized the sources of instruction level repetition and some characteristics of function-level behavior. They found that a large number of dynamic function calls are called with repeated arguments, and that most of these calls had either implicit inputs or side effects. This lead them to conclude that few functions could be memoized. However, with SoftBulk, implicit inputs and side effects are easily coped with.

Connors et al ([19, 20]) studied memoization of coarse-grained regions of code using a compiler (augmented with profiling information) to identify profitable regions. During execution, compiler-inserted instructions direct the hardware to record the explicit inputs and outputs for a region in a hardware table. Then, when the region is encountered again, the table is checked for a solution. If one exists, the outputs are written into registers directly by the hardware, and the region is skipped. To account for memory inputs, each table entry has a memory valid bit which is cleared anytime a memory input for that entry is potentially updated. The compiler is responsible for scheduling invalidate instructions in the code.

Wu et al ([96]) combine speculation and memoization to exploit more region-level reuse. Their central observation is that CRB entries are often conservatively invalidated. These entries can be speculatively presumed to be correct and reused while a speculative thread validates them in the background.

MemoiSE differs from all of these in that it only targets functions as opposed to arbitrary regions of code, which accounts for it detecting less redundancy. MemoiSE does have an advantage, in particular over [20], in that it can detect dynamically any memory accesses that invalidate an entry in the solution table. In addition, the memory accesses of a function do not need to be analyzed statically for correct memoizaton, but such analysis is useful if it can eliminate some functions from consideration. MemoiSE incurs overhead for table lookups which are done in hardware in [20], but if MemoiSE were used in conjunction with such a table, the overheads could be significantly reduced. MemoiSE, like [96], merges memoizaton and speculative execution; however, the application of this merging is quite different. MemoiSE leverages a checkpoint to further specialize the code and eliminate the overheads of table lookup, whereas [96] speculates to overcome lost memoization opportunity.

Ding and Li ([24]) propose a compiler directed memoization scheme implemented fully

in software. The compiler identifies coarse-grained regions of code for reuse, then generates the necessary code to store the inputs in the hash table, and check the hash table for memoized calls on future occurences. The compiler must prove that all inputs are invariant for a memoized region. Also, because there is no hardware support, the compiler must perform a cost-benefit analysis to decide when a region of code is worth memoizing. MemoiSE is similar to this approach in that the lookup table is a software structure and the compiler/profiler must decide which functions to transform using a similar cost analysis. MemoiSE, however, can more aggressively select functions since implicit inputs and outputs are checked dynamically.

4.8 Summary

SoftBulk is an architecture for exposing signatures and signature operations directly to software. SoftBulk enables software directed collection, disambiguation, and operations on signatures by providing a set extensions to the ISA.

To show the potential for SoftBulk, a novel memoization algorithm, MemoiSE, is proposed that leverages SoftBulk to record which memory locations are read and written into a signature and performs disambiguation on that signature. As a result, MemoiSE can easily test whether implicit inputs or side effects have been changed since the memoized call. MemoiSE reduced instruction count, on average, by 8% for the applications considered, with a maximum savings of 17%.

SoftBulk can also be used for many other optimizations and in support of other systems. Several proposals for RTD-based optimizations can be revisited in light of SoftBulk with potentially new application or more general use [3, 44, 61, 80]. Also, aggressive speculative optimizations based on checkpointing, as in [57], may benefit from SoftBulk's ability to record information about a program's dependences from before a checkpoint. Of course, SoftBulk can integrate into environments that already use signatures ([8, 14, 97]) to enhance the software's or programmer's control over signature building and disambiguation.

References

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," in *MICRO 36: Proceedings of the 36th International Symposium on Microarchitecture*, IEEE Computer Society, November 2003.

[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded Transactional Memory," in *Proceedings of the 11th International Symposium on High Performance Computer Architecture*, IEEE Computer Society, February 2005.

[3] D. Bernstein, D. Cohen, and D. E. Maydan, "Dynamic Memory Disambiguation for Array References," in *MICRO 27: Proceedings of the 27th Annual International Symposium on Microarchitecture*, ACM Press, 1994.

[4] A. Bhowmik and M. Franklin, "A General Compiler Framework for Speculative Multithreading," in *Proceedings of 14th ACM Symposium on Parallel Algorithms and Architectures*, ACM Press, August 2002.

[5] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 11, July 1970.

[6] D. Boggs, A. Baktha, J. Hawkins, D. Marr, J. A. Miller, P. Roussel, R. Singhal, B. Toll, and K. Venkatraman, "The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology," *Intel Technology Journal*, vol. 8, February 2004.

[7] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a Framework for Architectural-Level Power Analysis and Optimizations," in *ISCA'00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, ACM Press, June 2000.

[8] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun, "An effective hybrid transactional memory system with strong isolation guarantees," in *ISCA'07:Proceedings of the 34th Annual International Symposium on Computer Architecture*, ACM Press, June 2007.

[9] J. Casmira and D. Grunwald, "Dynamic Instruction Scheduling Slack," in *KoolChips Workshop held in conjunction with MICRO'00*, IEEE Computer Society, December 2000.

[10] L. Ceze, K. Strauss, J. Tuck, J. Renau, and J. Torrellas, "CAVA: Hiding L2 Misses with Checkpoint Assisted VAlue Prediction," *IEEE Computer Architecture Letters*, December 2004.

[11] L. Ceze, K. Strauss, J. Tuck, J. Renau, and J. Torrellas, "CAVA: Using Checkpoint-Assisted VAlue Prediction to Hide L2 Misses," *ACM Transactions on Architecture and Code Optimization*, June 2006.

[12] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: Bulk Enforcement of Sequential Consistency," in *ISCA'07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, ACM Press, June 2007.

[13] L. Ceze, J. Tuck, and J. Torrellas, "Are We Ready for High Memory-Level Parallelism?," in *4th Workshop on Memory Performance Issues*, IEEE Computer Society, February 2006.

[14] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval, "Bulk Disambiguation of Speculative Threads in Multiprocessors," in *ISCA '06: Proceedings of the 33rd Annual International Symposium on Computer Architecture*, IEEE Computer Society, 2006.

[15] L.-L. Chen and Y. Wu, "Aggressive Compiler Optimization and Parallelization with Thread-level Speculation," in *Proceedings of the International Conference on Parallel Processing*, IEEE Computer Society, October 2003.

[16] M. K. Chen and K. Olukotun, "The Jrpm System for Dynamically Parallelizing Java Programs," in *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, ACM Press, 2003.

[17] P. S. Chen, M. Y. Hung, Y. S. Hwang, R. D. Ju, and J. K. Lee, "Compiler Support for Speculative Multithreading Architecture with Probabilistic Points-to Analysis," in *Proceedings of the 2003 Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, ACM Press, June 2003.

[18] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture Optimizations for Memory-Level Parallelism," in *ISCA'04: Proceedings of the 31st International Symposium on Computer Architecture*, ACM Computer Society, June 2004.

[19] D. A. Connors, H. C. Hunter, B.-C. Cheng, and W. mei W. Hwu, "Hardware Support for Dynamic Management of Compiler-Directed Computation Reuse," in *ASPLOS IX: In the Proceedings of Architectural Support for Programming Languages and Operating Systems*, ACM Press, 2000.

[20] D. A. Connors and W. W. Hwu, "Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results," in *MICRO 32: Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, IEEE Computer Society, 1999.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 2001.

[22] Cray Computer, "U.S. Patent 6,665,774," December 2003.

[23] A. Cristal, D. Ortega, J. Llosa, and M. Valero, "Out-of-Order Commit Processors," in *HPCA'04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, IEEE Computer Society, February 2004.

[24] Y. Ding and Z. Li, "A Compiler Scheme for Reusing Intermediate Computation Results," in *CGO '04: Proceedings of the international symposium on Code generation and optimization*, IEEE Computer Society, 2004.

[25] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai, "A Cost-Driven Compilation Framework for Speculative Parallelization of Sequential Programs," in *Proceedings of SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, ACM Press, June 2004.

[26] P. Dubey, K. O'Brien, K. O'Brien, and C. Barton, "Single-Program Speculative Multithreading (SPSM) Architecture," in *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, 1995.

[27] T. Dunigan, J. Vetter, J. White, and P. Worley, "Performance Evaluation of the Cray X1 Distributed Shared-Memory Architecture," in *IEEE Micro Magazine*, January/February 2005.

[28] K. I. Farkas and N. P. Jouppi, "Complexity/Performance Tradeoffs with Non-Blocking Loads," in *ISCA'94: Proceedings of the 21st International Symposium on Computer Architecture*, ACM Press, April 1994.

[29] B. Fields, R. Bodik, and M. D. Hill, "Slack: Maximizing Performance Under Technological Constraints," in *International Symposium on Computer Architecture*, IEEE Computer Society, 2002.

[30] B. Fields, S. Rubin, and R. Bodik, "Focusing Processor Policies via Critical-Path Prediction," in *ISCA'01: Proceedings of the 28th International Symposium on Computer Architecture*, ACM Press, 2001.

[31] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu, "Dynamic Memory Disambiguation Using the Memory Conflict Buffer," *SIGOPS Operating Systems Review*, vol. 28, no. 5, 1994.

[32] A. Gandhi, H. Akkary, R. Rajwar, S. T. Srinivasan, and K. Lai, "Scalable Load and Store Processing in Latency Tolerant Processors," in *ISCA'05: Proceedings of the 32nd International Symposium on Computer Architecture*, ACM Press, June 2005.

[33] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun, "The Stanford Hydra CMP," *IEEE Micro*, vol. 20, no. 2, 2000.

[34] L. Hammond, M. Willey, and K. Olukotun, "Data Speculation Support for a Chip Multiprocessor," in *ASPLOS VIII: Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM Press, October 1998.

[35] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional Memory Coherence and Consistency," in *ISCA'04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, ACM Press, 2004.

[36] J. Huang and D. J. Lilja, "Exploiting Basic Block Value Locality with Block Reuse," in *HPCA'99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, IEEE Computer Society, 1999.

[37] T. Johnson, R. Eigenmann, and T. Vijaykumar, "Min-cut Program Decomposition for Thread-Level Speculation," in *Proceedings of SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, ACM Press, 2004.

[38] T. Juan, J. J. Navarro, and O. Temam, "Data Caches for Superscalar Processors," in *Proceedings of the 11th International Conference on Supercomputing*, IEEE Computer Society, July 1997.

[39] M. Kirman, N. Kirman, and J. F. Martinez, "Cherry-MP: Correctly Integrating Checkpointed Early Resource Recycling in Chip Multiprocessors," in *MICRO '05: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, IEEE Computer Society, 2005.

[40] N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martinez, "Checkpointed Early Load Retirement," in *HPCA'05: Proceedings of the 11th International Symposium on High Performance Computer Architecture*, IEEE Computer Society, February 2005.

[41] V. Krishnan and J. Torrellas, "A Chip-Multiprocessor Architecture with Speculative Multithreading," *IEEE Transactions on Computers*, September 1999.

[42] D. Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," in *ISCA'81: Proceedings of the 8th International Symposium on Computer Architecture*, ACM Press, May 1981.

[43] T. Li, A. R. Lebeck, and D. J. Sorin, "Quantifying Instruction Criticality for Shared Memory Multiprocessors," in *SPAA'03: Proceedings of the 15th Annual Conference on Parallel Algorithms and Architectures*, ACM Press, 2003.

[44] J. Lin, T. Chen, W.-C. Hsu, and P.-C. Yew, "Speculative Register Promotion Using Advanced Load Address Table (ALAT)," in *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*, IEEE Computer Society, 2003.

[45] M. Lipasti, C. Wilkerson, and J. Shen, "Value Locality and Load Value Prediction," in *ASPLOS VII: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM Press, October 1996.

[46] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," in *MICRO 29: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, IEEE Computer Society, 1996.

[47] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas, "POSH: A TLS Compiler that Exploits Program Structure," in *Proceedings of the 2006 Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*, ACM Press, March 2006.

[48] P. Marcuello and A. González, "Clustered Speculative Multithreaded Processors," in *ICS '99: Proceedings of the 13th International Conference on Supercomputing*, ACM Press, 1999.

[49] P. Marcuello and A. Gonzalez, "Thread-Spawning Schemes for Speculative Multithreading," in *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, IEEE Computer Society, February 2002.

[50] J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas, "Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors," in *MICRO 35: Proceedings of the 35th Annual International Symposium on Microarchitecture*, IEEE Computer Society, November 2002.

[51] J. F. Martinez and J. Torrellas, "Speculative Synchronization: Applying Thread-level Speculation to Explicitly Parallel Applications," in *ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM Press, 2002.

[52] D. Michie, ""Memo" Functions and Machine Learning," *Nature*, April 1968.

[53] K. Moore, J. Bobba, M. J. Moravam, M. Hill, and D. Wood, "LogTM: Log-based Transactional Memory," in *HPCA'06: Proceedings of the 12th International Symposium on High Performance Computer Architecture*, IEEE Computer Society, February 2006.

[54] A. Moshovos, G. Memik, A. Choudhary, and B. Falsafi, "JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers," in *HPCA'01: Proceedings of the 7th International Symposium on High Performance Computer Architecture*, IEEE Computer Society, January 2001.

[55] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," in *HPCA'03: Proceedings of the 9th International Symposium on High Performance Computer Architecture*, IEEE Computer Society, February 2003.

[56] R. Nagpal and A. Bhowmik, "Criticality Based Speculation Control for Speculative Multithreaded Architectures," in *ACM International Workshop on Advanced Parallel Processing Technologies*, ACM Press, 2005.

[57] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles, "Hardware Atomicity for Reliable Software Speculation," in *ISCA'07:Proceedings of the 34th Annual International Symposium on Computer Architecture*, ACM Press, 2007.

[58] A. Nicolau, "Run-Time Disambiguation: Coping with Statically Unpredictable Dependencies," *IEEE Transactions on Computers*, vol. 38, no. 5, 1989.

[59] J. T. Oplinger, D. L. Heine, and M. S. Lam, "In Search of Speculative Thread-Level Parallelism," in *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, IEEE Computer Society, October 1999.

[60] I. Park, C. L. Ooi, and T. N. Vijaykumar, "Reducing Design Complexity of the Load/Store Queue," in *MICRO 36: Proceedings of the 36th Annual International Symposium on Microarchitecture*, IEEE Computer Society, December 2003.

[61] M. Postiff, D. Greene, and T. N. Mudge, "The Store-load Address Table and Speculative Register Promotion," in *MICRO 32: Proceedings of the 32nd International Symposium on Microarchitecture*, IEEE Computer Society, 2000.

[62] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas, "Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization," in *ISCA'01: Proceedings of the 28th International Symposium on Computer Architecture*, ACM Press, June 2001.

[63] R. Rajwar and J. R. Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," in *MICRO 34: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, IEEE Computer Society, 2001.

[64] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing Transactional Memory," in *ISCA'05: Proceedings of the 32nd International Symposium on Computer Architecture*, ACM Press, June 2005.

[65] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC Simulator," January 2005. http://sesc.sourceforge.net.

[66] J. Renau, K. Strauss, L. Ceze, W. Liu, S. Sarangi, J. Tuck, and J. Torrellas, "Thread-Level Speculation On a CMP Can Be Energy Efficient," in *ICS'05: Proceedings of the 19th International Conference on Supercomputing*, ACM Press, 2005.

[67] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas, "Tasking with Out-of-order Spawn in TLS Chip Multiprocessors: Microarchitecture and Compilation," in *ICS'05: Proceedings of the 19th International Conference on Supercomputing*, ACM Press, 2005.

[68] P. Salverda and C. Zilles, "A Criticality Analysis of Clustering in Superscalar Processors," in *International Symposium on Microarchitecture*, IEEE Computer Society, 2005.

[69] S. Sastry, R. Bodik, and J. Smith, "Characterizing Coarse-grained Reuse of Computation," in *3rd ACM Workshop on Feedback-Directed and Dynamic Optmization, in conjuction with MICRO 33.*, ACM Press, 2000.

[70] C. Scheurich and M. Dubois, "The Design of a Lockup-free Cache for High-Performance Multiprocessors," in *ICS'88: Proceedings of the 2nd ACM/IEEE Conference on Supercomputing*, ACM Press, November 1988.

[71] J. S. Seng, E. S. Tune, and D. M. Tullsen, "Reducing Power with Dynamic Critical Path Information," in *MICRO 34: Proceedings of 34th Annual Symposium on Microarchitecture*, IEEE Computer Society, December 2001.

[72] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler, "Scalable Hardware Memory Disambiguation for High ILP Processors," in *MICRO 36: Proceedings of the 36th Annual International Symposium on Microarchitecture*, IEEE Computer Society, December 2003.

[73] P. Shivakumar and N. Jouppi, "CACTI 3.0: An integrated cache timing, power and area model," Tech. Rep. 2001/2, Compaq Computer Corporation, August 2001.

[74] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse," in *ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture*, ACM Press, 1997.

[75] A. Sodani and G. S. Sohi, "An Empirical Analysis of Instruction Repetition," in *ASPLOS-VIII: Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM Press, 1998.

[76] G. Sohi, S. Breach, and T. Vijayakumar, "Multiscalar Processors," in *ISCA'95: Proceedings of the 22nd International Symposium on Computer Architecture*, ACM Press, June 1995.

[77] G. Sohi and M. Franklin, "High-Bandwidth Data Memory Systems for Superscalar Processors," in *ISCA'91: Proceedings of the 18th Annual International Symposium on Computer Architecture*, ACM Press, May 1991.

[78] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual Flow Pipelines," in *ASPLOS XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM Press, October 2004.

[79] J. G. Steffan, C. Colohan, A. Zhai, and T. Mowry, "A Scalable Approach to Thread-Level Speculation," in *ISCA'00: Proceedings of the 27th International Symposium on Computer Architecture*, ACM Press, June 2000.

[80] B. Su, S. Habib, W. Zhao, J. Wang, and Y. Wu, "A Study of Pointer Aliasing for Software Pipelining Using Run-time Disambiguation," in *MICRO 27: Proceedings of the 27th Annual International Symposium on Microarchitecture*, ACM Press, 1994.

[81] L. Su and M. H. Lipasti, "Speculative optimization using hardware-monitored guarded regions for java virtual machines," in *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, ACM Press, 2007.

[82] D. Tarjan, S. Thoziyoor, and N. Jouppi, "CACTI 4.0," Tech. Rep. HPL-2006-86, Hewlett Packard Laboratories Palo Alto, June 2006.

[83] M. Tremblay, "MAJC: Microprocessor architecture for Java computing." Hot Chips, August 1999.

[84] J. Tsai, J. Huang, C. Amlo, D. Lilja, and P. Yew, "The Superthreaded Processor Architecture," *IEEE Transactions on Computers*, vol. 48, September 1999.

[85] J.-Y. Tsai, Z. Jiang, and P.-C. Yew, "Compiler techniques for the superthreaded architectures," *Int. J. Parallel Program.*, vol. 27, no. 1, 1999.

[86] J. Tuck, L. Ceze, and J. Torrellas, "Scalable Miss Handling Architectures for High MLP," in *MICRO 39: Proceedings of the 39th Annual International Symposium on Microarchitecture*, IEEE Computer Society, December 2006.

[87] J. Tuck, W. Liu, and J. Torrellas, "CAP: Criticality Analysis for Power-Efficienct Speculative Multithreading," in *ICCD'07: Proceedings of the International Conference on Computer Design*, IEEE Computer Society, October 2007.

[88] D. M. Tullsen and J. A. Brown, "Handling Long-Latency Loads in a Simultaneous Multithreading Processor," in *MICRO 34: Proceedings of the 34th International Symposium on Microarchitecture*, IEEE Computer Society, December 2001.

[89] E. Tune, D. Liang, D. Tullsen, and B. Calder, "Dynamic Prediction of Critical Path Instructions," in *HPCA'01: Proceedings of the 7th International Symposium on High Performance Computer Architecture*, IEEE Computer Society, 2001.

[90] E. Tune, D. M. Tullsen, and B. Calder, "Quantifying Instruction Criticality," in *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*, IEEE Computer Society, 2002.

[91] T. Vijaykumar and G. Sohi, "Task Selection for a Multiscalar Processor," in *Proceedings of the 31th Annual International Symposium on Microarchitecture*, IEEE Computer Society, November 1998.

[92] T. N. Vijaykumar, "Compiling for the Multiscalar Architecture," Ph.D. dissertation, University of Winsconsin-Madison, 1998.

[93] H. S. Wang, X. P. Zhu, L. S. Peh, and S. Malik, "Orion: A Power-Performance Simulator for Interconnection Networks," in *MICRO 35: Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2002.

[94] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for Store-wait-free Multiprocessors," in *ISCA'07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, ACM Press, 2007.

[95] J. Whaley and C. Kozyrakis, "Heuristics for Profile-Driven Method-Level Speculative Parallelism," in *Proceedings of the 34th International Conference on Parallel Processing*, IEEE Computer Society, 2005.

[96] Y. Wu, D.-Y. Chen, and J. Fang, "Better Exploration of Region-level Value Locality with Integrated Computation Reuse and Value Prediction," in *ISCA '01: Proceedings of the 28th Annual International Symposium on Computer Architecture*, ACM Press, 2001.

[97] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "LogTM-SE: Decoupling Hardware Transactional Memory from Caches," in *HPCA'07: Proceedings of the 13th International Symposium on High Performance Computer Architecture*, IEEE Computer Society, 2007.

[98] A. Zhai, C. Colohan, J. Steffan, and T. Mowry, "Compiler Optimization of Scalar Value Communication Between Speculative Threads," in *ASPLOS X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM Press, October 2002.

[99] H. Zhou and T. Conte, "Enhancing Memory Level Parallelism via Recovery-Free Value Prediction," in *ICS'03: Proceedings of the 17th International Conference on Supercomputing*, ACM Press, June 2003.

[100] C. B. Zilles and G. S. Sohi, "Master/Slave Speculative Parallelization," in *MICRO 35: Proceedings of the 35th Annual ACM/IEEE Symposium on Microarchitecture*, IEEE Computer Society, 2002.

Author's Biography

James Murray Tuck, III, was born in Mobile, AL, in 1978, and spent most of his childhood in Pell City, AL. James received his B.E., *summa cum laude*, from Vanderbilt University in December 1999, and he received his M.S. and Ph.D. from University of Illinois at Urbana-Champaign in December 2003 and October 2007, respectively, under the tutelage of Prof. Josep Torrellas. James' overall research focus is in computer architecture and compiler design, with the main focus on chip multiprocessors (CMPs) and hardware and software supports for aggressive speculative execution. While working on the Ph.D., James published more than ten articles in computer architecture, and he received an IEEE Micro Top Picks Paper Award in 2006 for his work in Thread-Level Speculation. James is a member of Tau Beta Phi, the IEEE Computer Society, and the ACM. After graduation, James joined the faculty at North Carolina State University as an Assistant Professor in Electrical and Computer Engineering.